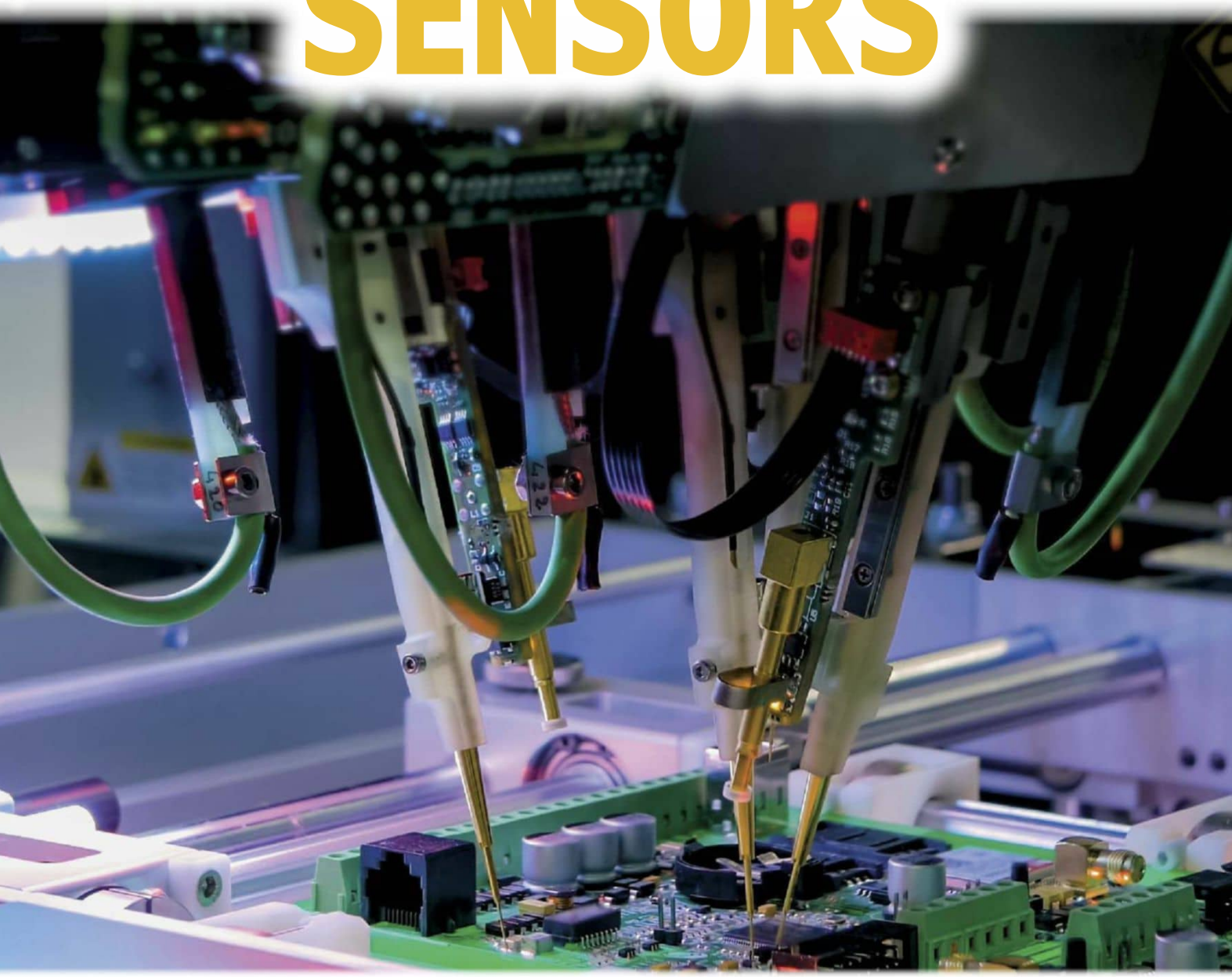
circuitcellar.com

SENSORS
DECEMBER 2025
ISSUE 425

circuit cellar

Inspiring the Evolution of Embedded Design

SENSORS

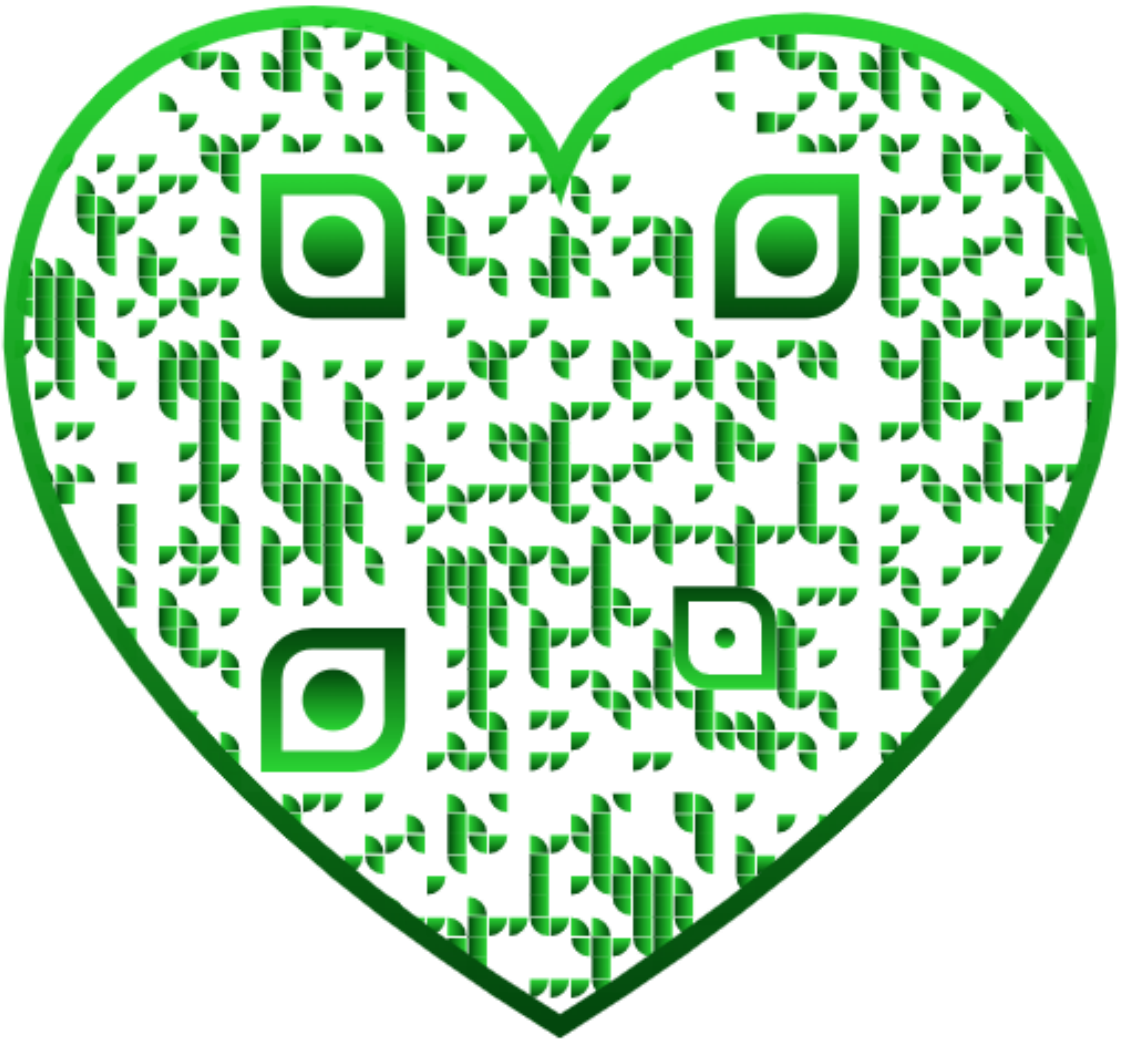


$9.00US $10.00CAN

12

0 74470 75349 0

Small Powerful Microcontrollers | Data Acquisition via Instruments Bus | Embedded DSP with the RT1010 MCU | Signal Conditioning with Op Amps | Encrypted MQTT Protocol | Embedded Software Architectures That Last | MCU Timing — What Actually Happens Inside That Chip | Embedded in Thin Slices Wrap Up | The Future of Sensors in Safety Systems



Imagine what you will

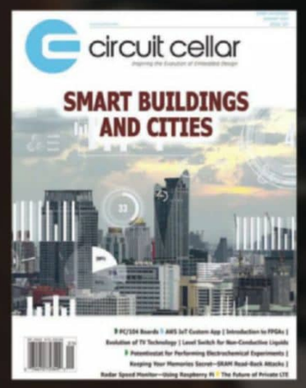
Discover

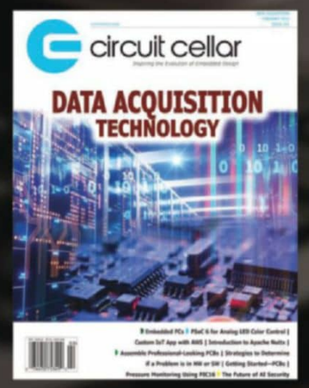
400+ ISSUES

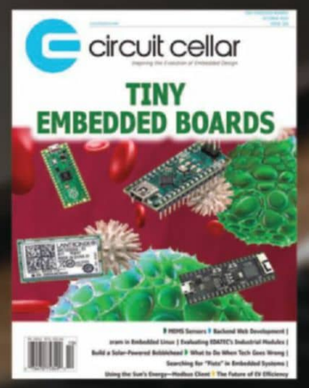
CC VAULT

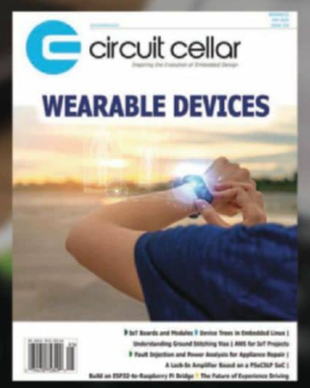
Get the complete *Circuit Cellar* issue archive and article code stored on a stylish, durable and portable USB flash drive. You can easily keep your CC Vault archive up to date by purchasing subsequent issues from our webshop or by downloading issues with a *Circuit Cellar* Digital Subscription. Issues appear in searchable PDF format.

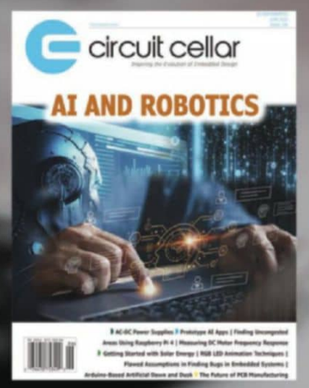
cc-webshop.com

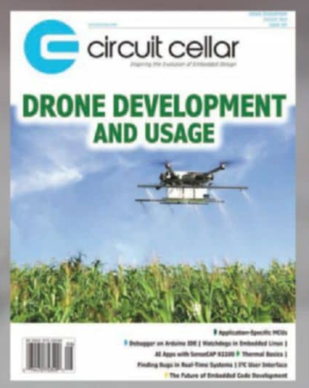


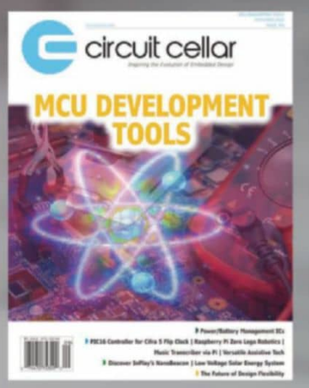












Complete archive includes PDFs of all issues in print through date of purchase.

INPUT Voltage

Excited for 2026

Getting ready for a new set of projects

I've just spent three days meeting engineers, embedded system designers, and "makers," and I have to tell you I am excited about the world of embedded system design today. There are a couple of major factors that spark that excitement, and I'll share some details with you that take it to the next level.

First, on the spurring items, we have a new class of single-board computers (SBCs), such as the Arduino Q, that combine microcontrollers and microprocessors on the same board. I'm looking forward to getting hands on with the Arduino Q and the new Arduino Lab IDE because the ability to write for both processors and integrate the programs in a single IDE is just mind-blowing.

I'm not alone in that thought—I spoke with several people working on robotics, scientific instrumentation, and process control who can't wait to get comfortable with the new class of dual-processor boards for projects they have in the works.

And, of course, new SBCs aren't the only things that are driving new applications and new projects. Mesh networking, such as that enabled by the LoRa protocol and its network Meshtastic, are allowing campus-wide and area-wide applications that communicate and control themselves without cabling. I spoke with many engineers and makers who are using Meshtastic in applications and had conversations using Meshtastic and a radio that I wore clipped to my badge lanyard.

I'm a big proponent of using appropriate radio technology as a control layer. I'm a radio amateur (KG4GWA, for those keeping score) and I'm finding that Meshtastic is leading me in some interesting directions as I plan projects that involve multiple sensors scattered around a site. My career has been long enough to let me experience a number of excitement waves around emerging technologies, but it's still nice to feel that burst of energy that leads to thinking about creative new projects.

Speaking of projects, we keep them rolling with this issue of *Circuit Cellar*. Tam Hanna brings us a project built around the Solartron 7150 while Raul Torrico takes us into digital signal processing with the first part of an article series using the RT1010 MCU.

We're continuing our series on designing analog circuits in one article while we read about MQTT and how to encrypt its traffic in another. All of these (plus our columns) are showing how our writers are pushing forward with technology and helping readers do the same.

And on the topic of columns, this is the issue where we say goodbye to Bob Japenga as a *Circuit Cellar* columnist. Bob is retiring to a well-deserved period of, I don't know, doing what he wants to do. I don't know how he'll survive without deadlines, but all of us at *Circuit Cellar* wish Bob the best and hope he knows that he's always welcome in our pages.

While we're talking about columns, 2026 will see some new columns in our pages. Look for new columns focusing on sensors, RF in embedded applications, and Linux starting soon.

My question this month is this: Are you looking forward to working with the new generation of dual-processor SBCs? Does one have you more excited than another? Let me know! You can see how to reach me at the links below—I'm looking forward to the conversation.

Curtis Franklin
Editor-In-Chief
cfranklin@circuitcellar.com
LinkedIn: https://www.linkedin.com/in/curtisfranklin/
Facebook: https://www.facebook.com/curtis.f.franklin
Instagram: https://www.instagram.com/curt_franklin/
Mastodon: https://mastodon.sdf.org/@Kg4gwa
Bluesky: https://bsky.app/kg4gwa.bsky.social

Issue 425 December 2025 | ISSN 1528-0608

CIRCUIT CELLAR® (ISSN 1528-0608) is published monthly by:

KCK Media Corp.
PO Box 417, Chase City, VA 23924

Periodical rates paid at Chase City, VA, and additional offices. One-year (12 issues) subscription rate US and possessions $50, Canada $65, Foreign/ ROW $75. All subscription orders payable in US funds only via Visa, MasterCard, international postal money order, or check drawn on US bank.

SUBSCRIPTION MANAGEMENT

Online Account Management: circuitcellar.com/account
Renew | Change Address/E-mail | Check Status

CUSTOMER SERVICE

E-mail: customerservice@circuitcellar.com
Phone: 434.533.0246
Mail: Circuit Cellar, PO Box 417, Chase City, VA 23924

Postmaster: Send address changes to
Circuit Cellar, PO Box 417, Chase City, VA 23924

NEW SUBSCRIPTIONS

circuitcellar.com/subscription

ADVERTISING

Contact: Hugh Heinsohn
Phone: 757-525-3677
Fax: 888-980-1303
E-mail: hheinsohn@circuitcellar.com
Advertising rates and terms available on request.

NEW PRODUCTS

Contact: Kirsten Campbell
E-mail: product-editor@circuitcellar.com

HEAD OFFICE

KCK Media Corp.
PO Box 417
Chase City, VA 23924
Phone: 434-533-0246

COPYRIGHT NOTICE

Entire contents copyright © 2025 by KCK Media Corp. All rights reserved. Circuit Cellar is a registered trademark of KCK Media Corp. Reproduction of this publication in whole or in part without written consent from KCK Media Corp. is prohibited.

DISCLAIMER

KCK Media Corp. makes no warranties and assumes no responsibility or liability of any kind for errors in these programs or schematics or for the consequences of any such errors printed in Circuit Cellar®. Furthermore, because of possible variation in the quality and condition of materials and workmanship of reader-assembled projects, KCK Media Corp. disclaims any responsibility for the safe and proper function of reader-assembled projects based upon or from plans, descriptions, or information published in Circuit Cellar®.

The information provided in Circuit Cellar® by KCK Media Corp. is for educational purposes. KCK Media Corp. makes no claims or warrants that readers have a right to build things based upon these ideas under patent or other relevant intellectual property law in their jurisdiction, or that readers have a right to construct or operate any of the devices described herein under the relevant patent or other intellectual property law of the reader's jurisdiction. The reader assumes any risk of infringement liability for constructing or operating such devices.

© KCK Media Corp. 2025 Printed in the United States

OUR NETWORK

circuit cellar · LinuxGizmos · audioxpress · VOICE COIL · LIS LOUDSPEAKER INDUSTRY SOURCEBOOK

SUPPORTING COMPANIES

BusBoard Prototype Systems	31
EmbeddedTS	61, C4
Embedded World 2026	21
Front Panel Express	9
RECOM	37
SeaFire Micros, Inc.	C3
Siborg Systems, Inc.	25

NOT A SUPPORTING COMPANY YET?
Contact Hugh Heinsohn
hugh@circuitcellar.com,
Phone: 757-525-3677,
Fax: 888-980-1303
to reserve space in the
next issue of *Circuit Cellar.*

THE TEAM

FOUNDER	Steve Ciarcia
PUBLISHER	KC Prescott
CONTROLLER	Chuck Fellows
EDITOR-IN-CHIEF	Curtis Franklin
CONTRIBUTING EDITOR	Brian Millier
CONTENT MANAGER- EU	Jurij Mikeln
SENIOR ASSOCIATE EDITOR	Shannon Becker
TECHNICAL COPY EDITOR	Carol Bower
PROJECT EDITORS	Ken Davidson
	David Tweed
MARKETING MANAGER	Tori Zienka
ADVERTISING COORDINATOR	Diana Winsor
ADVERTISING SALES REP.	Hugh Heinsohn

COLUMNISTS
Stuart Ball (Start to Finish)
Jacob Beningo (Modern Embedded Insights)
Bob Japenga (Embedded in Thin Slices)
Brian Millier (Picking Up Mixed Signals)
Colin O'Flynn (Embedded Systems Essentials)

FEATURES

4 **Data Acquisition with the General Purpose Instrument Bus**
Part 1: Design and Installation
By Tam Hanna

10 **Embedded DSP with the RT1010 MCU**
Part 1: Workflow and FIR Filters
By Raul Alvarez-Torrico

22 **Designing Analog Electronics**
Part 2: Signal Conditioning with Op Amps
By Andrew Levido

29 **Encrypted MQTT Protocol for Critical Sectors**
Mechanisms, Challenges, and Best Practices
By Ignacio Siccardi

32 TECHNOLOGY FEATURE
Sensors in the Spotlight
The Next Decade of Embedded Sensor Systems
By Curtis Franklin

38 DATASHEET
Small Size, Big Power
Smaller Microcontrollers Bring New Possibilities
By Curtis Franklin

COLUMNS

42 Modern Embedded Insights
Designing Embedded Software Architectures That Last
By Jacob Beningo

48 Start to Finish
Understanding MCU Timing
What Actually Happens Inside That Chip?
By Stuart Ball

54 Embedded in Thin Slices
Bob's Wrap Up
By Bob Japenga

63 TECH THE FUTURE
The Future of Sensors in Safety Systems
Sensing the Stop
How Magnetic Sensors Are Enabling the Next Generation of Braking Systems
By Karsten Köhler

58 : PRODUCT NEWS
62 : TEST YOUR EQ

@editor_cc @circuitcellar · circuitcellar · circuitcellar

Data Acquisition with the General Purpose Instrument Bus

Part 1: Design and Installation

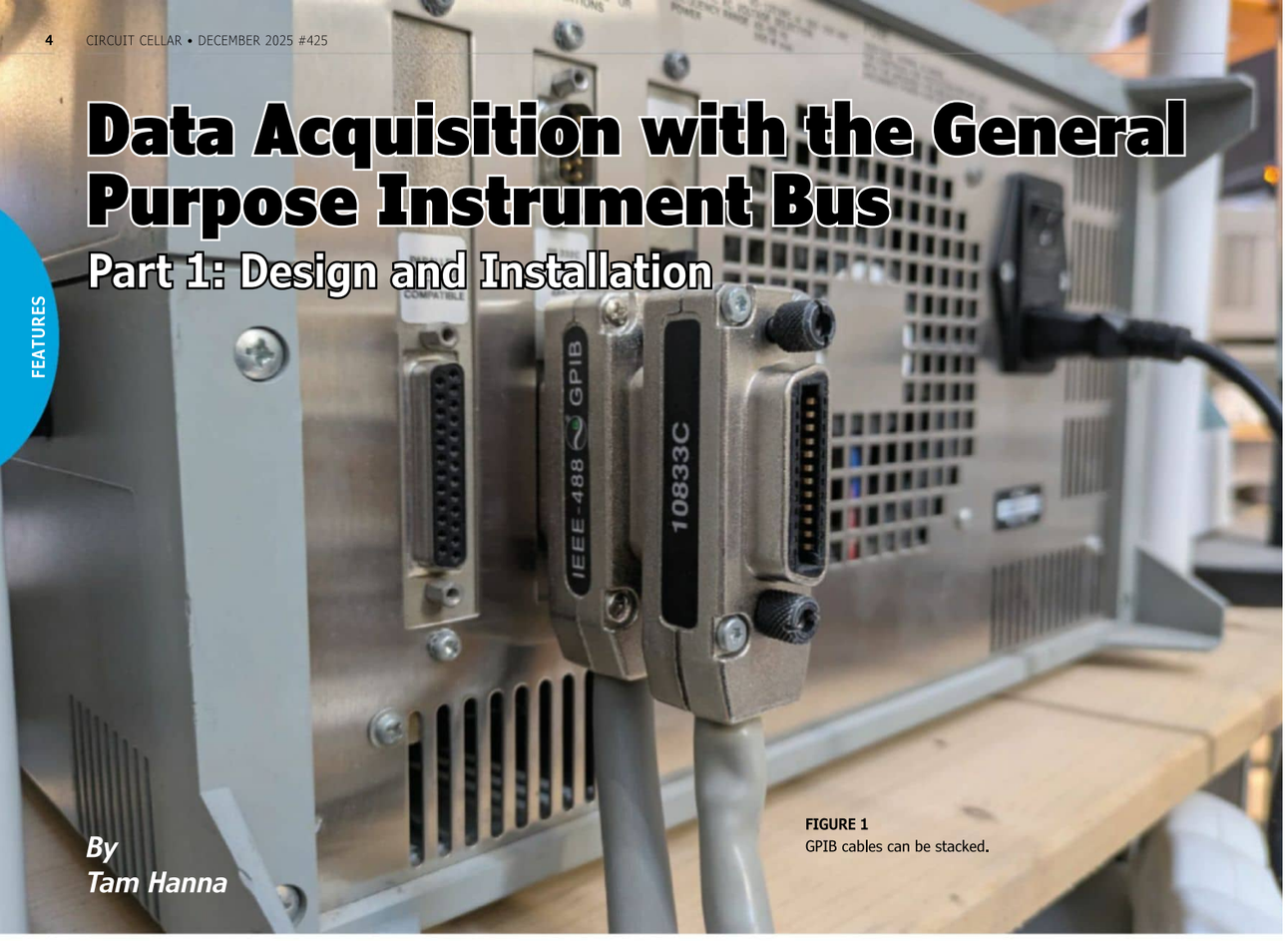


FIGURE 1
GPIB cables can be stacked.

By
Tam Hanna

The General Purpose Instrument Bus is used to link test and measurement instruments for exchange of data and commands. In Part 1 of this article series, Tam steps us through the basic considerations for designing and installing a system.

The General-Purpose Instrument Bus (GPIB), specified initially by Hewlett-Packard as HP-IB, is a standard interface found at the back of most test equipment. It is a parallel digital communication interface for connecting and controlling test and measurement instruments, which can link computers and instruments from different manufacturers to exchange data and commands. Thanks to the used equipment market, home laboratories can interact with this technology affordably.

GPIB differs from modern computer networking standards by having a relatively chaotic bus architecture. Cable connectors can be stacked on top of one another (**Figure 1**). This way, multiple devices can be connected in a tree-like structure. This is not unorthodox; in practice, keep cable weight in mind as GPIB cables connected back-to-back put strain on the PCB of the test equipment.

The standard permits up to 15 connected devices at the same time, and the maximum cable length should be less than 20m. In practical laboratories, the primary constraint is the number of active devices. The claim is that half of the connected devices must be active at all times. I ignored this often, usually without adverse results. However, if weird behavior occurs, switching on additional devices might be a good first step.

SELECTING GPIB HARDWARE

The following steps assume the presence of GPIB-based test equipment. We will assume that the goal is creating a data processing pipeline, with the data feeding algorithmic processes such as those in automated test equipment (ATE) software.

Should your task be to acquire screenshots, do not follow the instructions outlined in this article. Instead, purchase one or more KISS-488 units [1]. These are GPIB-to-Ethernet dongles with virtual printer firmware. They attach to the back of the test equipment that needs to print and provide results via a web page.

GPIB is an Institute of Electrical and Electronics Engineers (IEEE) Standard. In practice, the best results are achieved with a product from NI (now Emerson) [2]. In this case, the least compatibility problems occur with the Windows/.NET environment. As of Linux kernel version 6.17, the Kernel developer team plans to include the GPIB module as an official module [3.]

Given that the following steps take place on a workstation, we use a NI PCIe-to-GPIB card. The used market provides the large-format version for approximately €200. I saw one toil for almost 10 years next to an AMD Thousand Islands GPU.

However, the documentation for the first-generation card claims maximum ambient temperatures of 0 to 35°C. This should be kept in mind when using the card in a workstation with high case temperature. Multiple inquiries to NI, seeking clarification on whether the 35°C value refers to room or case temperature, went unanswered. NI lists the differences between the two card versions in "Notice of Board Change—NI PCIe-GPIB" [4].

Other interfaces, such as an ExpressCard-to-GPIB or a USB-to-GPIB dongle, can also be found. When purchasing the latter, take extreme care, because the design of the GPIB-to-USB converter is often faked.

Due to their sturdy nature, GPIB cables are virtually indestructible and show up at industrial surplus auctions. Chinese vendors on AliExpress also offer these cables but be sure to test them before the return window slams shut.

THE WINDOWS DEVELOPMENT ENVIRONMENT

The following steps outline the Windows development environment, which includes C# and Visual Studio.

NI explicitly requires that any development environment intended for use with GPIB hardware must be configured before the NI installer runs. For experimentation, install both Visual Studio 2022 and the newly released Visual Studio 2026. In both cases, select the .NET desktop development payload.

After all updates have been deployed successfully, reboot the Windows workstation and download the drivers from the NI website [5]. Creating and logging into an Emerson account is required to download drivers and other software from the site. In the following steps, we will use version NI-488.2 2025 Q4.

When installing on a single computer, click "Download" to open the online installer named ni-488.2_25.8_online.exe. If multiple machines must be updated, hit the "install offline" link. It downloads a larger file that can be deployed without additional component downloads.

Next, run the .exe file. Make sure to include the various programming examples for Visual Studio 2010, 2012, and 2013, along with all other components. Only the NI-VISA server and the NI Linux RT system images are unnecessary. Accept the license agreements to start the deployment. Given that some software needs to be downloaded, expect a slight wait.

THE SOLARTRON 7150 MULTIMETER

During installation, an analysis of the various methods for determining the GPIB address of the individual devices is in order. For the Solartron 7150 Multimeter [6] used in the following steps, a bank of DIP switches at the back of the instrument configures the transceiver.

1) Correctness can be checked during power-up, as the instrument will display its communication address. DIP switches degrade over time, with often unpredictable results. If an instrument shows weird behavior, investigating the mechanics is a good first step to a successful repair.

Utility für die Hardwarekonfiguration
Datei Bearbeiten Ansicht Werkzeuge Hilfe
DESKTOP-DC8H7UJ
DESKTOP-DC8H7UJ System Product Name PC System Serial...
GPIB0 NI PCIe-GPIB Gerät zur Messgerätesteuerung 013217AC
Instrument 0 NI-VISA-GPIB-Gerät
Keine Auswahl
Wählen Sie zur Anzeige der Einstellungen ein Objekt.

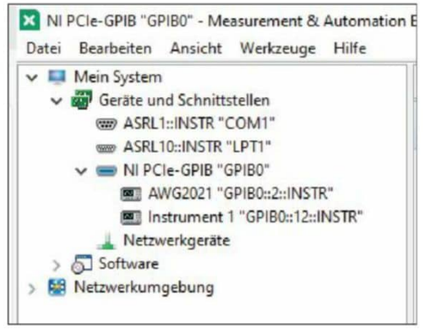
FIGURE 2
Our GPIB card was detected successfully.

2) Modern equipment handles this task via firmware. Either way, note down the GPIB address of the GPIB equipment for the following steps. In my network, the Multimeter lives at address 12. After that, the Emerson Installer should have finished its duty. A reboot is required to give Windows the opportunity to update its internal caches.
3) After the reboot, NI will present a registration dialog, which can be closed without consequences. When using the new software version, launch the NI Hardware configuration utility to begin the process of the GPIB network initialization. It presents the GPIB Card in a fashion similar to the one shown in **Figure 2**.
4) Next, the communication must be verified using NI Max. The GPIB Card can be expanded and selected as shown in **Figure 3.** The tree structure shows the devices that were successfully detected.
5) In the case of the Solartron 7150, a special concern must be kept in mind. Unlike the Danaher Tektronix AWG2021, the device is not able to fully identify itself to a GPIB host. This is caused by the lack of the firmware's capability to answer the IDN command, leading to the behavior shown in **Figure 4**. The IDN command and the rest of the SCPI issue are discussed later.
6) For now, a second "gotcha" concerns scanning for instruments. Observing the toolbar, we find a "Refresh" button and a "Scan for instruments" button.

   The list of GPIB Instruments connected is refreshed only if the "Scan for instruments" button is pressed. This must be kept in mind at all times, since GPIB is not able to perform automatic discovery of newly connected devices.
7) Next, start Visual Studio 2026, and use its project generator to generate a new project based on the "WPF-App (.NET Framework)" template. Note: Do not select "WPF-Application," because this leads to a non-working project skeleton when combined with the NI488 code. The various driver libraries work best when used in WPF and cannot exist when used in UWP and similar runtimes.
8) Importantly, select .NET 4.5.2 or 4.8 as the target framework version. The Emerson libraries do not support more modern versions of the .NET Framework as of the writing of this article.
9) Next, right-click the References folder in the Project Explorer to open the "Add Reference" menu. Only if the .NET Framework 4.x is used, the dialogue will provide an "Assemblies" section in the left sector of the window.

   Next, switch to the "Extensions" tab and select both "National Instruments 488.2" and "NationalInstruments.Common assemblies." Incidentally, the 488.2 library will have spaces within the string, whereas the common libraries file name makes do without the spaces. This is not a printing error in *Circuit Cellar*.
10) Click the "OK" button to close the reference window and integrate the newly added assemblies into the project's compilation tree.
11) After that, a first test program can be run. This is shown in **Listing 1**. First, instances of the classes `NationalInstruments.NI4882.Board` and `NationalInstruments.NI4882.Devices` are needed. The first represents the GPIB card, whereas the second represents the multimeter.
12) We first populate the GPIB card variable by simply selecting the first member of the Board selector. The examples in this article assume that only one GPIB card is present. After that, we use the address mentioned earlier to create an instance of the multimeter object. Given that both constructors directly interact with hardware, we check for an exception. If it does not occur, we can assume that the device connection was successful.
13 Run the program and verify that the string "GPIB OK!" appears in the output console.

FIGURE 3
This card found a total of two devices.



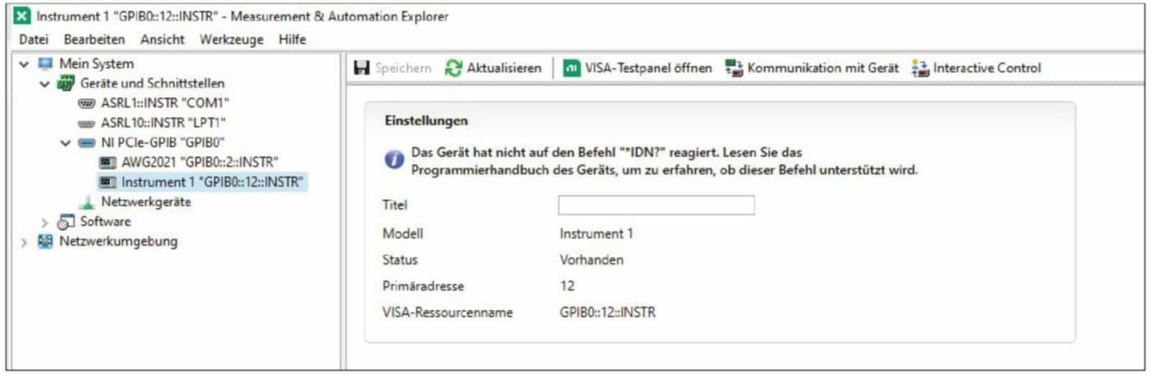


FIGURE 4
NI Max complains about limited firmware capabilities.

HARVESTING INFORMATION

At this point, data collection can begin. Most GPIB-based instruments support two modes: triggered and continuous. In continuous mode, the instrument permanently acquires information, which is then held in a round-robin buffer. In triggered mode, the system starts acquisition only when commanded to do so.

Each of these modes has its advantages: If data acquisition should begin only after a specified condition exists (for example, power supply–DUT-multimeter interaction), the triggered mode is preferable. For us, continuous mode is easier, because it provides an endless stream of information to process.

Recent GPIB-based instruments implement the "SCPI standard." Its best-known part is the earlier-mentioned IDN string, which NI Max uses for self-identification. SCPI, the Standard Commands for Programmable Instruments, specifies a string-based format for metrology-related messages, along with a few generic, supportive commands. These are, by and large, human-readable. **Listing 2** shows a sequence that transfers an arbitrary waveform from the workstation to the above-mentioned Danaher Tektronix AWG2021.

For our Solartron 7150, a simpler command set should be used. It is fully documented in the multimeter's manual [7].

In particular, our Solartron multimeter expects a command string to be transmitted. With this information, we can proceed to run the sample program shown in **Listing 3**. For simplicity, only the new code is printed; the rest can be found in Listing 1.

The new version of the code first invokes the `Write` method to send the format string to the multimeter. Its job is to set various parameters, such as accuracy, measurement mode, and the way triggering occurs.

An invocation of `Thread.Sleep` gives the test equipment time to adjust its internal configuration. Keep in mind that many pieces use attenuators or signal routes based on electromechanical systems, so switching them takes time. After that, the `ReadString` method is invoked to read any available information from the GPIB device. It is then emitted to the Visual Studio output console, where it appears in a format similar to the one shown in **Figure 5**.

A careful study of the format strings supported by the Solartron multimeter reveals that its embedded processor can also operate in a special mode that combines multiple readings into a single one. Adjusting

```
public partial class MainWindow : Window {
  NationalInstruments.NI4882.Board myGPIBCard;
  NationalInstruments.NI4882.Device myMultimeter;
  public MainWindow() {
    InitializeComponent();
    try     {
      myGPIBCard = new NationalInstruments.NI4882.Board(0);
      myMultimeter = new Device(0, 12);
      Console.WriteLine("GPIB OK!");
    }
    catch (NationalInstruments.NI4882.GpibException e) {
      Console.WriteLine("ISSUE: GPIB error! " + e.Message);
    }
  }
}
```

LISTING 1
A first test program checks the communication between the GPIB card and multimeter.

```
        myGPIBCard = new NationalInstruments.NI4882.Board(0);
        myDanaher = new Device(0, 2);
        myDanaher.Write(":DEBUG:SNOOP:STATE ON");
        myDanaher.Write(":DATA:INIT");
        myDanaher.Write(":DATA:DEST \"LECROY.WFM\"");
        myDanaher.Write(":DATA:WIDTH 1");
        myDanaher.Write(":CURVE #3255");
        for (int i = 0; i < 256; i++) myDanaher.Write(new
byte[] { (byte) i });
```

LISTING 2
How SCPI transfers look.

```
    public MainWindow() {
      InitializeComponent();
      try     {
        myGPIBCard = new NationalInstruments.NI4882.Board(0);
        myMultimeter = new Device(0, 12);
        myMultimeter.Write("MOR3ION1T1\n");
        Thread.Sleep(300);
        String myStr = myMultimeter.ReadString();
        Console.WriteLine(myStr);
```

LISTING 3
Starting to read data from our Solartron multimeter.

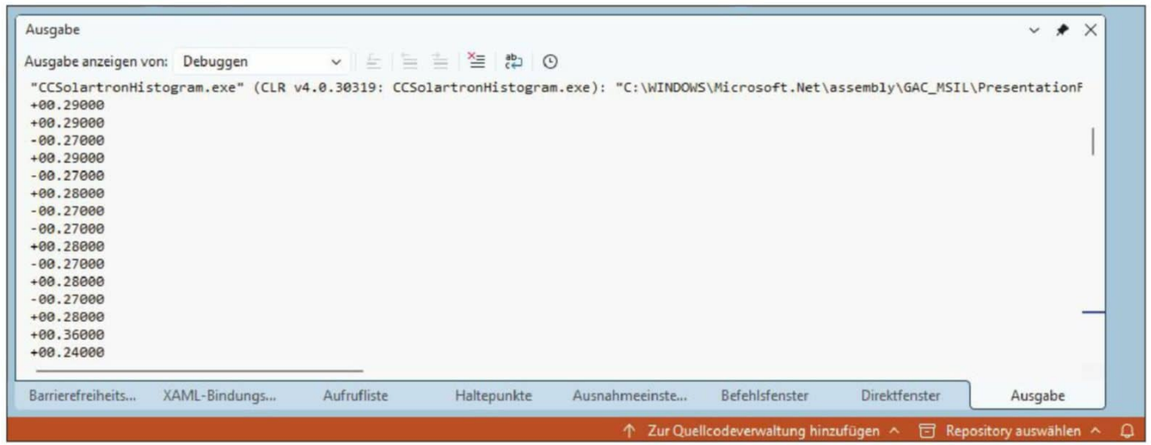


FIGURE 5
The first bit of measurement information is available on the workstation.

the format string (like `myMultimeter.Write("M0R3I4N1T1\n");`) and giving more computation time (code `Thread.Sleep(5000);`) should be enough to achieve a working program. In practice, this is not the case; instead, the console displays the error "ISSUE: GPIB error! Time limit exceeded to complete operation."

This error is caused by the `ReadString` method, which expects a relatively large amount of information and fails otherwise. Sadly, our filtered invocation provides relatively little data and takes a long time to do so. Even if timed perfectly, the string would be too short to satisfy the buffer, leading to the timeout error.

This kind of interaction between the Visual Studio code and the test equipment happens commonly when dealing with GPIB test equipment. In many cases, a brute-force solution can be found. One classic example is the reading method shown in **Listing 4.**

```
public String readSolartron() {
  String myStr = "";
  try {
    while (1 == 1) {
      myMultimeter.IOTimeout = TimeoutValue.T100ms;
      String x = myMultimeter.ReadString(1);
      myStr = myStr + x;
    }
  }
  catch (Exception) { }
  return myStr;
```

LISTING 4
Reading information from a Solartron multimeter via a brute-force approach.

In this listing, `readSolartron()` starts by reducing the timeout. After that, an endless loop keeps reading individual bits from the multimeter and collates them in a string. The idea is that the infinite loop gets terminated by an exception occurring. After that, the method considers its task done, and returns the collected information. When used instead of the NI-provided "Read" code, an output similar what is shown in **Figure 6** is achieved. It is clearly visible that the filtered string shows up on the workstation, along with the usual complaints about the raised exception.

FROM DATA TO NUMERICAL REPRESENTATION

Microsoft's .NET Framework integrates itself closely with the machine used to execute code. This can have interesting consequences when converting data. As you will see in the next test, we will try to convert our acquired data to a numeric format. A simple approach would use the Parse method of the float primitive:

```
Console.WriteLine(myStr);
float x = float.Parse(myStr);
Console.WriteLine(x);
```

Unfortunately, running this does not yield a satisfactory result. In the command line, the output will be structured as follows:

```
-00.00014
-14
```

The reason for this problem is Solartron's unique format: 0 values come with a prefix of 00, while the wrong comma delimiter is used. Both of these problems can be worked around with string manipulation:

```
myStr = myStr.Replace("00.", "0,");
myStr = myStr.Replace(".", ",");
float x = float.Parse(myStr);
```

Running this version of the code yields the following, correct output:

```
+00.00004
4E-05
```

Now, an important note: When working on ATE software, do not be afraid to use software "shims" to reduce the amount of time actual test equipment needs to stay powered on. Caching and replaying responses from a text file or a string resource is a classic and effective technique. Alternatively, filter modes and similar slowdowns can be disabled to reduce waiting time during algorithm development.

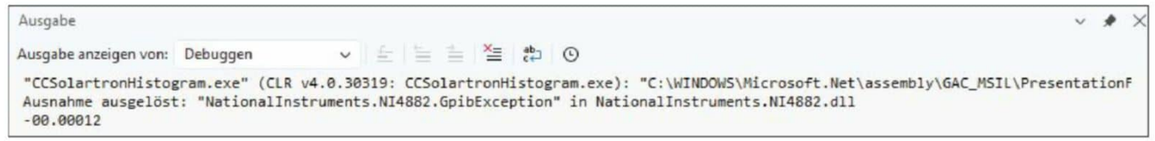


FIGURE 6
The filtered value is visible.

NEXT TIME: CONQUER CHARTING WITH SCOTTPLOT

GPIB application development appears to be an interesting challenge, due to the collision of various technology generations. In practice, most issues can be worked around with a bit of glue code; once the ATE system works, it is a significant time saver.

However, our journey through the world of ATE is not finished. Now that data acquired from the Solartron 7150 is available in a numerical format, it can be converted into a variety of representations. Stay tuned for the next part; the fun is not over, yet!

ABOUT THE AUTHOR

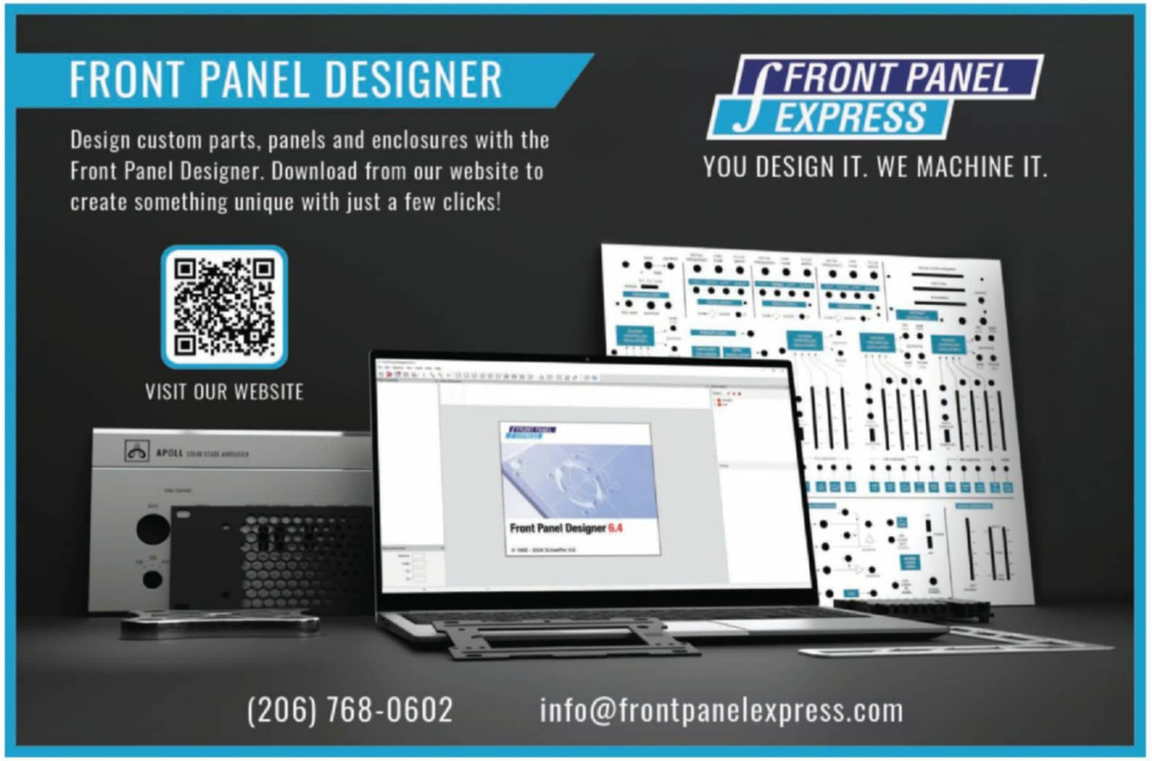
Tam Hanna began his career writing assembly programs for the PIC16F84A. After a hiatus, during which he focused on mobile phone software development, he eventually returned to electronics. In his spare time, the electrical engineer enjoys 3D printing, restoring test equipment, writing scientific books, and smoking cigars.

Additional materials from the author are available at: www.circuitcellar.com/article-materials

References [1] to [7] as marked in the article can be found there.

RESOURCES

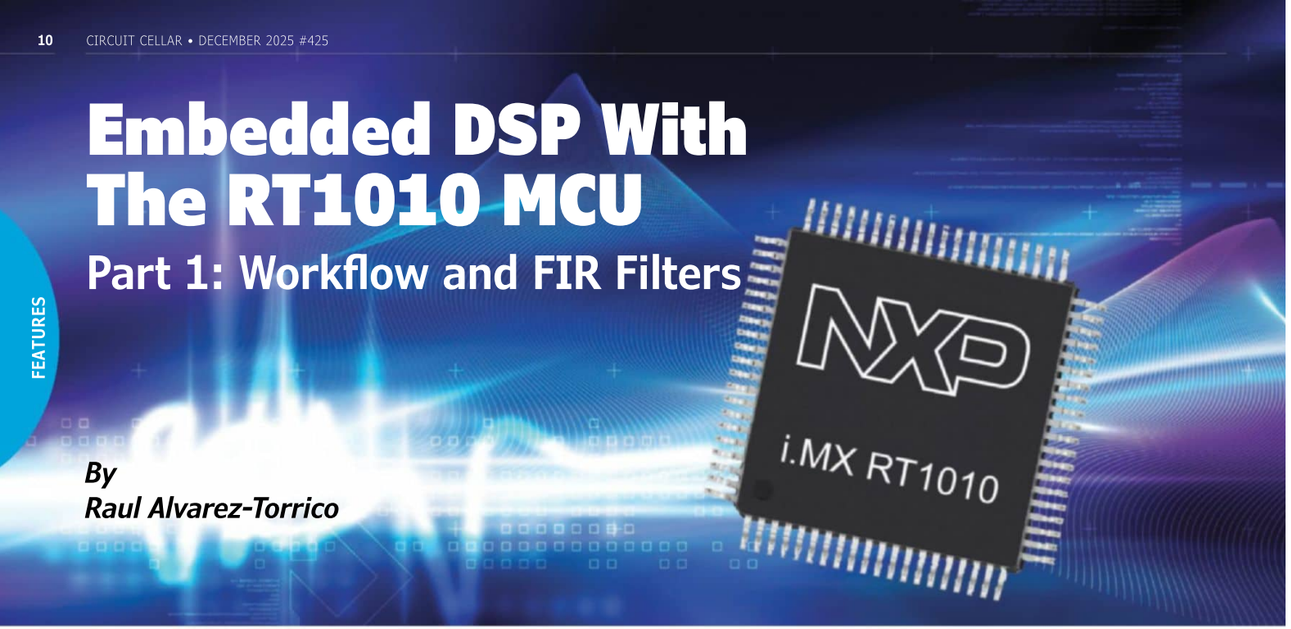
National Instruments | www.ni.com


FRONT PANEL DESIGNER
Design custom parts, panels and enclosures with the Front Panel Designer. Download from our website to create something unique with just a few clicks!
FRONT PANEL EXPRESS
YOU DESIGN IT. WE MACHINE IT.
VISIT OUR WEBSITE
Front Panel Designer 6.4
(206) 768-0602
info@frontpanelexpress.com

FEATURES

Embedded DSP With The RT1010 MCU

Part 1: Workflow and FIR Filters



By
Raul Alvarez-Torrico

In DSP, a digital audio filter processes and modifies audio frequency content and phase. In Part 1 of this series, Raul uses an NXP RT1010 Evaluation Kit to create a simple example and workflow, and then uses a finite impulse response (FIR) filter to process audio in real time.

This article series presents a framework for developing embedded audio digital signal processing (DSP) applications using the NXP i.MX RT1010 Evaluation Kit (MIMXRT1010-EVK). In Part 1, I will build upon an example of digital audio record-playback from the board's software development kit (SDK) to establish a processing workflow using the C programming language. I will then describe the implementation of finite impulse response (FIR) filters, which are essential components in audio equalizers and other audio filtering applications.

Next, I will introduce a simple control interface incorporating a push button and a potentiometer to enable or bypass the filter and adjust the audio output gain.

In Part 2 of the series, I will explore time-delay-based effects, modulation, and distortion, while enhancing the control interface to support better real-time filter parameter adjustments.

This series is aimed at embedded systems developers and enthusiasts eager to explore audio DSP. To follow along, readers should have experience with embedded systems development using ARM microcontrollers (MCUs), ideally within the NXP ecosystem, or similar MCUs and IDEs from other manufacturers. A basic understanding of analog-to-digital (ADC) conversion and digital audio representation is also recommended.

A DIGITAL AUDIO FILTER

In Digital Signal Processing (DSP), a digital audio filter is a computational algorithm that processes and modifies audio frequency content and phase. Filtering, and audio processing in general, can serve technical purposes, such as noise reduction and audio restoration, or artistic and aesthetic goals, including frequency shaping, equalization, modulation, distortion, and other effects. To perform DSP, audio must first be converted from the analog domain through a process called analog-to-digital conversion (ADC). After processing in the digital domain, the audio is converted back to analog using the reverse process, known as digital-to-analog conversion (DAC).

Audio DSP can be done using fixed-point or floating-point arithmetic. Fixed-point processing employs integer and fractional arithmetic with 16-bit, 24-bit, or 32-bit integer data types. In contrast, floating-point processing uses 32-bit or 64-bit floating-point data types.

Fixed-point processing is computationally efficient, offering faster execution with lower latency, making it ideal for real-time audio applications. However, it typically provides a limited dynamic range. It is well-suited for battery-powered devices, due to its lower power consumption, but it is more prone to audio signal overflow, underflow, and sample clipping distortion. Additionally, its reduced precision can introduce quantization errors, particularly in recursive algorithms such as IIR filters, Fast Fourier Transform (FFT), or convolution. Rounding errors may also accumulate, contributing to noise.

Alternatively, floating-point arithmetic offers a broader dynamic range, reducing the risks of overflow, underflow, and clipping. It provides greater precision, minimizing quantization errors in demanding algorithms such as FFT, IIR, and convolution, making it well suited for high-fidelity audio processing. However, floating-point operations are computationally more complex and slower than fixed-point operations. They consume more processing time and energy, which can reduce efficiency for real-time audio applications on resource-constrained devices such as MCUs. Additionally, floating-point formats are not immune to numerical computation issues, such as rounding errors, or the occurrence of "infinite" or "not-a-number" (NaN) values resulting from calculations.

Floating-point DSP is therefore generally practical for embedded systems only when the MCU includes a hardware Floating Point Unit (FPU), such as the one found in the i.MX RT1010 MCU on the MIMXRT1010-EVK Board.

I.MX RT1010 AUDIO DSP PRIMER

The MIMXRT1010-EVK development board (**Figure 1**) features audio peripherals ideal for experimenting with DSP. It includes a Wolfson Microelectronics WM8960 stereo codec with 24-bit sigma-delta ADCs and DACs, supporting high-quality oversampling conversion. The codec offers three configurable stereo input sources for flexible audio routing. Additionally, it integrates a microphone preamplifier, a headphone amplifier, and a Class D stereo speaker amplifier, delivering 1W per channel into 8Ω loads using a 5V power supply.

Complementing the WM8960 codec, the MIMXRT1010 board incorporates a condenser electret microphone and a 3.5mm Tip-Ring-Ring-Sleeve (TRRS) audio jack, which includes a secondary microphone input and stereo headphone output. Both the electret microphone and audio jack connect directly to the WM8960 codec. We will be using these peripherals to develop and test our DSP code.

To compile code for the i.MX RT1010, first install NXP's MCUXpresso IDE and the corresponding MIMXRT1010 board SDK package. To verify your setup, use the IDE's "Import SDK Examples" menu, search for "sai," and explore the driver_examples directory. Specifically, the "sai_edma_record_playback" example captures audio from both the onboard microphone and the input jack, then outputs it as a stereo signal through the headphone jack. We will build upon this example to establish our DSP workflow.

SIMPLE PLAYBACK

Listing 1 presents code for a simple audio processing example derived from the sai_edma_record_playback example in the MIMXRT1010 SDK. It reads a batch of stereo audio samples from the codec, adjusts the audio amplitude (gain), and outputs the modified samples back to the codec. The output gain ramps from 0 to 100% over 0.25 seconds, then back to 0% over another 0.25 seconds, repeating this cycle continuously. This linear ramping up and down of the audio gain effectively modulates the amplitude with a triangular, low-frequency oscillation (LFO). Let's explore how this works.

For simplicity, we omit most boilerplate code from the sai_edma_record_playback example, which initializes the MCU and WM8960 codec. Lines 12–13 define the "RxBuffer" and "TxBuffer" two-dimensional (2D) arrays, replacing the single Buffer used in the original sai_edma_record_playback example. Double buffering is preferred in audio DSP to prevent dropouts, glitches, and over-run/under-run issues. Each 2D buffer contain four "sub-buffers," each 1,024 bytes. For audio processing, we will handle one sub-buffer block at a time from both RxBuffer and TxBuffer. These sub-buffers will be rotated using an index, making them effectively "ring buffers."

Both buffers use 8-bit unsigned integer types. Since we work with 16-bit stereo

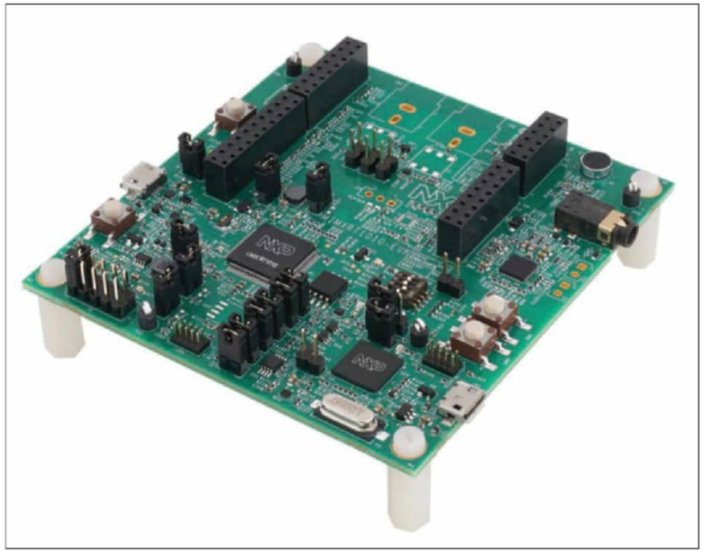


FIGURE 1
The MIMXRT1010-EVK board.

audio samples, each sub-buffer block stores 1,024 / (2 channels × 2 bytes) = 256 stereo audio samples per iteration. In the main() function, we first initialize the direct memory access (DMA), synchronous audio interface (SAI), system clocks, and WM8960 codec. This boilerplate code, largely drawn from the sai_edma_record_playback example, is omitted here for brevity. Within the infinite while loop, in line 23 we define the variable halfModCycle as:

```
halfModCycle = secs × (sample_rate
   / buffer_samples)
```

```
1 #include <stdio.h>
2 #include “app.h”
3 #include “board.h”
4 #include “fsl_debug_console.h”
5 #include “fsl_sai_edma.h”
6 #include “fsl_codec_common.h”
7
8 // Definitions
9 #define BUFFER_SIZE   (1024U)
10 #define BUFFER_NUMBER (4U)
11
12 AT_NONCACHEABLE_SECTION_ALIGN(static uint8_t RxBuffer[BUFFER_NUMBER][BUFFER_SIZE], 4);
13 AT_NONCACHEABLE_SECTION_ALIGN(static uint8_t TxBuffer[BUFFER_NUMBER][BUFFER_SIZE], 4);
14
15 int main(void) {
16     //...
17     BOARD_InitHardware();
18     PRINTF(“SAI example started!\n\r”);
19
20     // Init DMA, SAI, clocks, codec...
21
22     while (1) {
23         const float halfModCycle = 0.25 * (DEMO_AUDIO_SAMPLE_RATE / 256.0);
24         static float counter = 0.0;
25         static float inc = 1.0;
26
27         // Receive audio into RxBuffer[rx_index]
28         if (SAI_TransferReceiveEDMA(DEMO_SAI, &rxHandle, &(sai_transfer_t){
29                 .data = RxBuffer[rx_index],
30                 .dataSize = BUFFER_SIZE }) == kStatus_Success)
31         {
32             float gain = counter / halfModCycle; // 0.0 to 1.0
33             counter += inc;
34             if((counter >= halfModCycle) || (counter <= 0)) {
35                 inc = -inc;
36             }
37
38             // Apply gain and copy to TxBuffer
39             int16_t *src = (int16_t *)RxBuffer[rx_index];
40             int16_t *dst = (int16_t *)TxBuffer[tx_index];
41             for (uint32_t i = 0; i < BUFFER_SIZE / sizeof(int16_t); i++) {
42                 dst[i] = src[i] * gain;
43             }
44             rx_index = (rx_index + 1) % BUFFER_NUMBER;
45
46             // Queue TxBuffer for transmit
47             if (SAI_TransferSendEDMA(DEMO_SAI, &txHandle, &(sai_transfer_t){
48                     .data = TxBuffer[tx_index],
49                     .dataSize = BUFFER_SIZE }) == kStatus_Success)
50             {
51                 tx_index = (tx_index + 1) % BUFFER_NUMBER;
52             }
53         }
54     }
55 }
```

LISTING 1
Code for a simple audio processing example derived from the ai_edma_record_playback example in the MIMXRT1010 SDK..

This yields approximately 16, representing the number of processing iterations needed for the audio gain to ramp from 0 to 100% in 0.25 seconds and back to 0 in the same duration, resulting in an LFO modulation period of 0.5 seconds. The sample rate is set to 16,000Hz, but this can be changed during codec initialization.

Lines 28–30 invoke the SAI_TransferReceiveEDMA() function to receive BUFFER_SIZE (1,024) bytes into RxBuffer[rx_index]. If successful, the function returns kStatus_Success, allowing further processing. Line 32 calculates the gain that will be applied to all 256 stereo samples in the current batch. This gain will change from 0.0 to 1.0 over 16 processing iterations. When counter reaches 16, inc switches from 1.0 to -1.0, causing the gain to decrease gradually from 1.0 to 0.0 over the next 16 iterations. When counter reaches zero, inc becomes positive again for the next modulation cycle.

Line 39 declares a 16-bit integer source pointer src to RxBuffer at rx_index, to access the received codec data as 16-bit words. Line 40 similarly defines a destination pointer dst to TxBuffer. Lines 41–43 iterate from 0 to 511 to process 256 16-bit stereo audio samples. As shown in line 42, the processing is straightforward: an audio sample from RxBuffer, accessed via src, is multiplied by the current gain and stored in TxBuffer via dst. Left and right samples are interleaved, with src[0] containing the first left sample, src[1] the first right sample, src[2] the second left sample, and so forth. After processing the batch, line 44 increments rx_index by one and applies modulo BUFFER_NUMBER (4), resetting the index to 0 when it reaches 5. This enables sequential use of the four reception sub-buffer blocks.

Finally, we call the SAI_TransferSendEDMA() function to queue the TxBuffer[tx_index] sub-buffer for transfer to the codec (see lines 47–49). After queuing the block, we increment tx_index.

To test this example, compile and run the code on the MIMXRT1010 board. Place an audio source, such as a cell phone playing music, near the board's electret microphone, and connect headphones to the headphone jack. You should hear the audio output volume fluctuating in cycles of approximately 0.5 seconds.

DIGITAL FILTERS

Digital filters are used to amplify or attenuate energy at specific frequencies. Common types include low-pass, high-pass, band-pass, and band-reject (notch) filters. Regarding audio, these can be combined to create advanced processors, such as graphic and parametric equalizers. Digital filters are implemented using two primary mathematical algorithms: Finite Impulse Response (FIR) and Infinite Impulse Response (IIR).

FIR filters are non-recursive, feed-forward, and inherently stable, as they lack feedback loops. They can be designed to achieve a linear phase response, minimizing phase distortion, which is often critical in audio processing. FIR filters are straightforward to design and implement. They offer precise control over frequency response and predictable latency, making them ideal for real-time systems. They are used in equalizers where linear phase is essential. However, achieving sharp cutoffs or narrow bandwidths typically requires higher-order filters (more taps), increasing computational load and memory usage.

In contrast, IIR filters are recursive and may become unstable due to their use of feedback loops. They generally exhibit a non-linear phase response, which can introduce phase distortion. Designing IIR filters is more complex, and stability depends on carefully selected feedback coefficients and sufficient numerical precision. IIR filters are more computationally efficient, however, achieving sharp frequency responses with lower-order designs (fewer taps). They are well suited for notch and band-pass filters, and can emulate analog filter designs, such as Butterworth, Chebyshev, or Elliptic filters, but they offer less flexibility for arbitrary frequency responses.

FIR FILTERS

The block diagram and mathematical formula of a FIR filter is shown in **Figure 2**. A detailed explanation is beyond the scope of this article, but comprehensive descriptions can be found in any digital signal processing (DSP) textbook [1]. Nonetheless, the operating principle of an FIR filter is straightforward and can be intuitively understood.

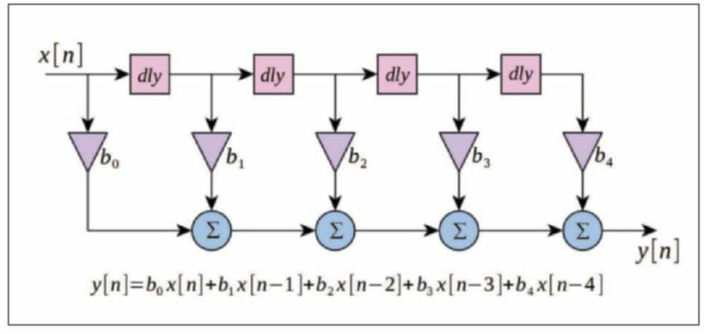


FIGURE 2
FIR filter block diagram and math formula.

FEATURES

Let's break down its mathematical equation. Suppose we've just read the fifth audio sample from the codec and aim to compute its corresponding filtered output. The equation would be expressed as:

$$y[4] = b0 \times x[4] + b1 \times x[3] + b2 \times x[2] + b3 \times x[1] + b4 \times x[0]$$

where:
y[4] is the filtered output corresponding to the current fifth sample x[4].
x[3]...x[0] are the samples read and processed before x[4].
b0 - b4 are the filter weighting coefficients.

As you can see, the current output y[4] depends not only on the current input x[4] but also on the four preceding samples. This configuration is known as a 4th-order filter with five "taps," where each tap is the product of a sample and its corresponding filter coefficient. Filters can have lower or higher orders, but higher-order filters typically achieve steeper cutoff responses.

The "dly" (delay) blocks in Figure 2 represent the time required to accumulate each subsequent sample. Consequently, FIR filters introduce a delay in the output signal, the duration of which depends on the filter's order. More taps enable sharper cutoffs, reduced ripple, better attenuation, precise frequency control, and improved phase linearity—but at the expense of increased output signal delay.

FIR FILTER IN PRACTICE

Implementing an FIR filter in code is straightforward once you have the filter coefficients. Many online and offline software tools are available for designing filters. Typically, these tools require you to specify the filter type (such as low-pass, high-pass, band-pass) and parameters such as frequency bands, gain/attenuation, sampling frequency, and filter order. The software then calculates the coefficients for you. For our examples, we will use the TFilter online tool [2], though many other online and offline options are available.

The TFilter homepage (**Figure 3**) showcases a five-tap, low-pass, FIR filter design. Upon opening the webpage, the graph is empty, and the parameter table (lower left) displays default values, which differ from those in Figure 3. The table's first row represents the first filter band, and the second row represents the second. I adjusted the default frequency values for both bands—the sampling frequency, and the number of taps. The filter is identified as low pass because the first (lower) band has a gain of 1, whereas the second (higher) band has a gain of 0. After configuring the parameters, clicking the "DESIGN FILTER" button (lower center) generates the filter coefficients shown in the right pane below the logo.

The "Gain vs. Frequency" graph illustrates the filter's calculated frequency response. For the first band, I specified a gain of 1 (100%)

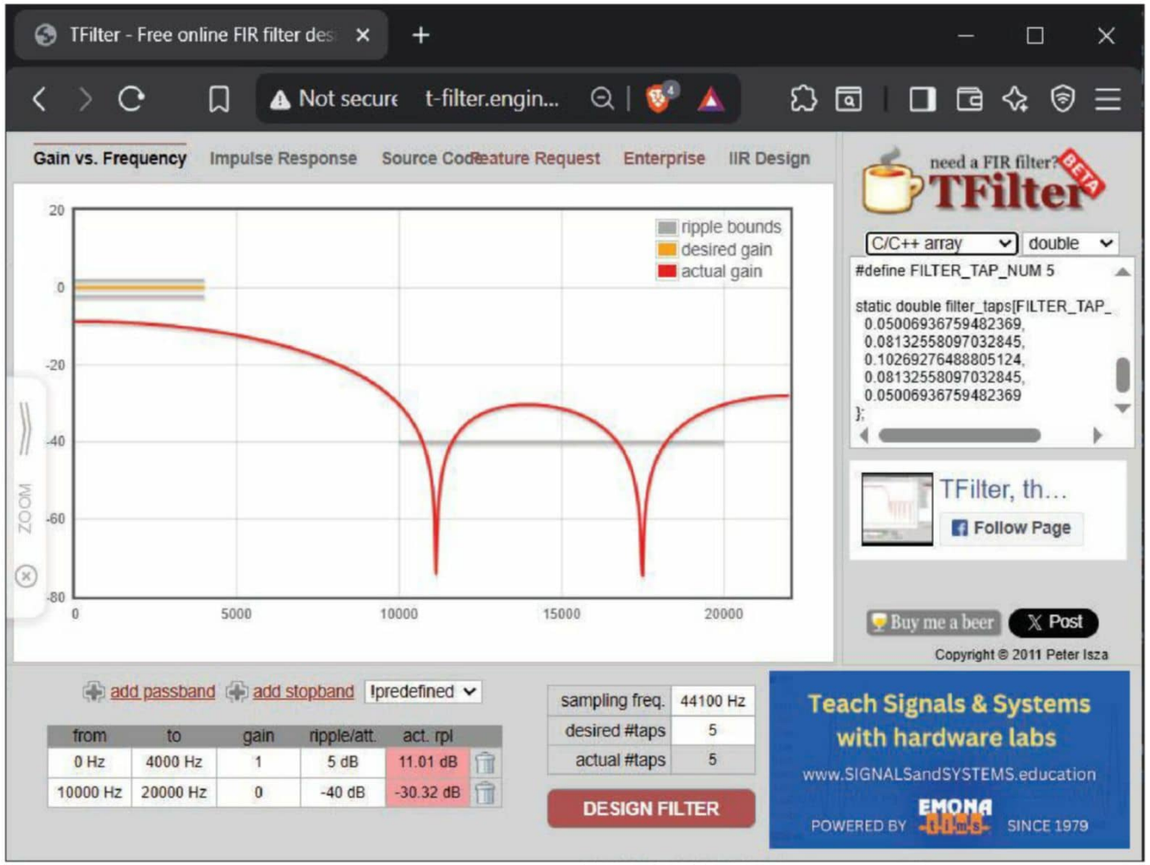


FIGURE 3
The TFilter five-tap, low-pass FIR filter design..

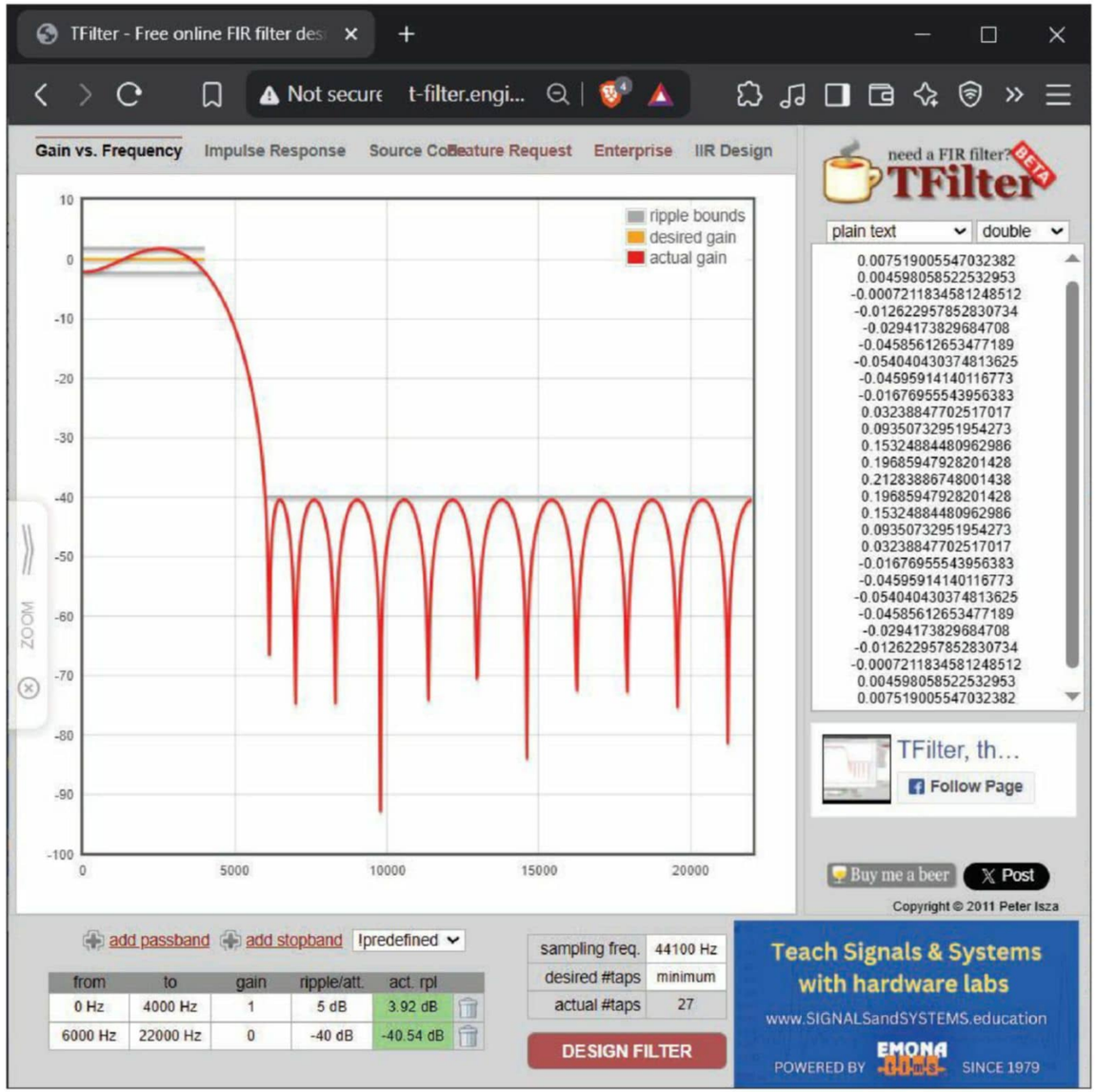


FIGURE 4
The design for a 27-tap (26th order), low-pass FIR filter..

```
// Other includes, globals & auxiliary function declarations...

#define FIR_TAP_NUM 27
const float firCoeffs[FIR_TAP_NUM] = {
  0.007519005547032382, 0.004598058522532953, -0.0007211834581248512,
  -0.012622957852830734, -0.0294173829684708, -0.04585612653477189,
  // ...
  -0.0007211834581248512, 0.004598058522532953, 0.007519005547032382
};

static float lpfHistLeft[FIR_TAP_NUM] = {0};
static float lpfHistRight[FIR_TAP_NUM] = {0};

float firFilter(float input, float *history, const float *coeffs) {
  for (int i = FIR_TAP_NUM - 1; i > 0; i--)
    history[i] = history[i - 1]; // Shift history

  history[0] = input;

  float result = 0;
  for (int i = 0; i < FIR_TAP_NUM; i++)
    result += coeffs[i] * history[i]; // Apply filter

  return result;
}

int main(void) {
  //...
  BOARD_InitHardware();
  PRINTF("SAI example started!\n\r");

  // Init DMA, SAI, clocks, codec...

  while (1) {
    // Receive audio into RxBuffer[rx_index]
    if (SAI_TransferReceiveEDMA(DEMO_SAI, &rxHandle, &(sai_transfer_t){
        .data = RxBuffer[rx_index],
        .dataSize = BUFFER_SIZE }) == kStatus_Success)
    {
      /* Apply filter and copy to TxBuffer */
      int16_t *src = (int16_t *)RxBuffer[rx_index];
      int16_t *dst = (int16_t *)TxBuffer[rx_index];
     uint32_t totalSamples = BUFFER_SIZE / sizeof(int16_t); // Stereo, so left/right pairs
      for (uint32_t i = 0; i < totalSamples; i += 2) {
        // Read from RxBuffer & convert 16-bit int samples to floats
        float leftIn = (float)src[i];
        float rightIn = (float)src[i + 1];

        // Apply filtering
        float leftOut = firFilter(leftIn, lpfHistLeft, firCoeffs);
        float rightOut = firFilter(rightIn, lpfHistRight, firCoeffs);
        // Convert from float to int and copy to TxBuffer
        dst[i]   = (int16_t)leftOut;
        dst[i + 1] = (int16_t)rightOut;
      }

      if (SAI_TransferSendEDMA(DEMO_SAI, &txHandle, &(sai_transfer_t){
          .data = TxBuffer[rx_index],
          .dataSize = BUFFER_SIZE }) == kStatus_Success)
      {
        rx_index = (rx_index + 1) % BUFFER_COUNT;
      }
    }
  }
}
```

LISTING 2
Modifications to the earlier "volume scaling" example to incorporate the 26th-order, low-pass FIR filter.

with a ripple of 5dB (see the parameter table in Figure 3). But the actual ripple was 11.01dB, and the graph shows a gain loss of approximately 9dB (where 0dB represents 100% gain). For the second band, I specified a gain of 0 with a ripple of -40dB, but the actual ripple reached -30.32dB. This suboptimal performance results from using only five taps. As we will explore next, increasing the number of taps by using a higher-order filter significantly improves the response.

An alternative filter design is shown in **Figure 4**. Unlike the previous one, I selected "minimum" for "desired #taps" (see the lower center region of Figure 4), instead of specifying a fixed number. This allows the software to determine the minimum number of taps automatically, while meeting all other constraints. The right pane of Figure 4 contains the 27 computed filter coefficients for the resulting 26th-order FIR filter. As you can see, with this design we achieved equal or better ripple attenuation than specified.

How is this filter implemented in code? **Listing 2** presents modifications to the earlier "volume scaling" example to incorporate this filter. Lines 3–9 define the `firCoeffs` array, which contains the 27 filter coefficients, though most values are omitted for brevity. A download link to complete source files is available on the *Circuit Cellar* Article Code

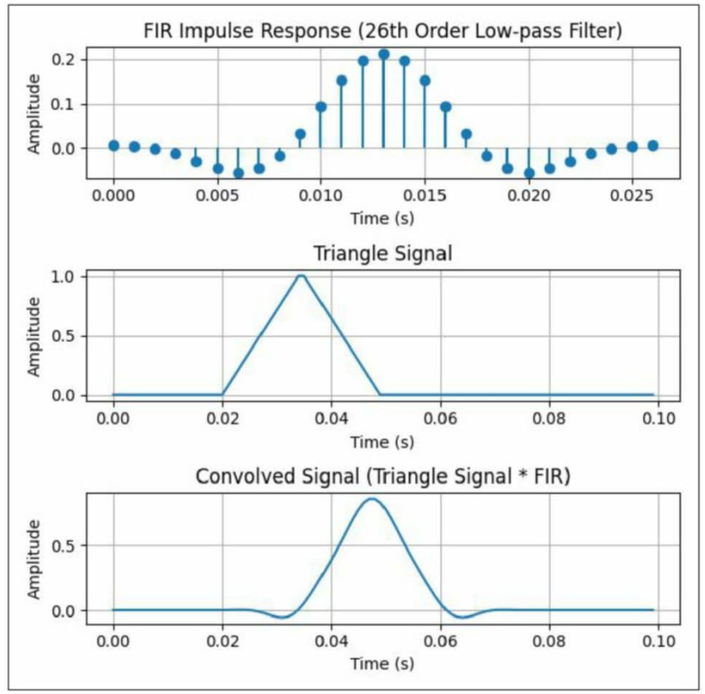


FIGURE 5
26th-order FIR filter convolution.

```
// Previous includes + push button & potentiometer includes...

// Push button & potentiometerglobal variables
volatile bool g_InputSignal = false;
bool g_FilterOn = true;
volatile bool g_AdcConversionDoneFlag;
volatile uint32_t g_AdcConversionValue;
volatile uint32_t g_PrevAdcConversionValue;

// Other globals and function prototypes...
// GPIO & ADC interrupt handlers
void EXAMPLE_GPIO_IRQHandler(void) {
  GPIO_PortClearInterruptFlags(EXAMPLE_SW_GPIO, 1U << EXAMPLE_SW_GPIO_PIN);
  g_InputSignal = true;
  SDK_ISR_EXIT_BARRIER;
}
void EXAMPLE_ADC_IRQHandler(void) {
  g_AdcConversionDoneFlag = true;
 g_AdcConversionValue = ADC_GetChannelConversionValue(DEMO_ADC_BASE, DEMO_ADC_CHANNEL_GROUP);
  SDK_ISR_EXIT_BARRIER;
}

int main(void) {
  // GPIO, ADC, interrupts & audio initialization...
  while (1) {
    if (g_InputSignal) {
```

LISTING 3
26th-order,low-pass FIR filter with control interface.

FEATURES

and Files webpage. Lines 11–12 declare two `history` arrays to store 27 input samples for each channel (left and right).

Lines 14–25 define the `firFilter()` function, which performs the filter computation. Lines 15–16 shift the previous input samples in the `history` buffer to the right, discarding the oldest sample at `history[FIR_TAP_NUM - 1]` to make room for the new input sample, passed as a parameter and stored in `history[0]` (line 18). After updating the buffer, the filter is applied to the samples. Lines 21–22 iterate over the `history` samples and coefficients, performing a one-to-one multiplication and accumulating the products in the `result` variable. The function then returns this variable as the filtered value for the current input sample.

Within the `while` infinite loop (lines 34-65), we first receive a batch of 1,024 bytes (lines 36-38). Next, we declare a 16-bit integer pointer `src` to the `RxBuffer` and a `dst` pointer to the `TxBuffer` (lines 41-42). We then iterate over both buffers 256 times (lines 44-55) to apply the FIR filter to the left and right channels separately. Unlike the previous amplitude modulation example, where the same gain was applied to both channels (all 512 samples) without distinguishing among them, here the channels are processed independently, each with its own history buffer, since they may have different content.

```
27        delay();
28        if (1 == GPIO_PinRead(EXAMPLE_SW_GPIO, EXAMPLE_SW_GPIO_PIN)) {
29          g_FilterOn = !g_FilterOn;
30        }
31        g_InputSignal = false; /* Reset state of switch. */
32      }
33
34  ADC_SetChannelConfig(DEMO_ADC_BASE, DEMO_ADC_CHANNEL_GROUP, &adcChannelConfigStruct);
35     float gain = 1.0; // 100% audio volume
36     uint16_t delta = abs(g_AdcConversionValue - g_PrevAdcConversionValue);
37
38    if (g_AdcConversionDoneFlag && (delta >= 100 )) {
39      g_AdcConversionDoneFlag = false;
40      gain = (float)g_AdcConversionValue / 4095.0;
41      g_PrevAdcConversionValue = g_AdcConversionValue;
42    }
43
44      if (SAI_TransferReceiveEDMA(...) == kStatus_Success) {
45        int16_t *src = (int16_t *)RxBuffer[rx_index];
46        int16_t *dst = (int16_t *)TxBuffer[rx_index];
47       uint32_t totalSamples = BUFFER_SIZE / sizeof(int16_t); // Stereo, so left/right pairs
48
49        for (uint32_t i = 0; i < totalSamples; i += 2) {
50        if(g_FilterOn) {
51          float leftIn = (float)src[i];
52          float rightIn = (float)src[i + 1];
53          float leftOut = fir_filter(leftIn, lpfHistLeft, firCoeffs);
54          float rightOut = fir_filter(rightIn, lpfHistRight, firCoeffs);
55          dst[i]   = (int16_t)leftOut * gain;
56          dst[i + 1] = (int16_t)rightOut * gain;
57        } else {
58          dst[i]   = src[i];
59          dst[i + 1] = src[i + 1];
60        }
61        }
62        if (SAI_TransferSendEDMA(...) == kStatus_Success) { ... }
63      }
64    }
65  }
66  static void delay(void) {
67    volatile uint32_t i = 0;
68    for (i = 0; i < 1000000; ++i) { __NOP(); }
69  }
```

LISTING 3
Continued

Accordingly, on line 46, we read a 16-bit sample from `src[i]` (left channel), and on line 47, we read another 16-bit sample from `src[i + 1]` (right channel). This reflects the interleaved nature of left and right channel samples in the codec's data stream. Both samples are converted from integers to floats. Line 50 applies the FIR filter to the left sample by calling the `firFilter` function, passing the left sample, the left history buffer, and the filter coefficients as function arguments. Line 51 performs the same operation for the right sample. Lines 53-54 then convert the filtered results back to 16-bit integers and store them in the `TxBuffer`, pointed to by `dst`, maintaining the interleaved order. Lines 57-59 queue the `TxBuffer` to transfer the processed samples back to the codec.

Mathematically, a FIR filter generates its output through a convolution operation between the input signal "x" and the filter's impulse response, defined by its coefficients. Convolution is a fundamental and widely used technique in signal processing. Technically, an FIR filter is a convolution filter. It is commonly employed in audio DSP to create a diverse range of audio synthesis and effects, including room and space simulation (reverb), speaker and amplifier emulation, instrument and sound coloration, convolution-based distortion and saturation, time-domain effects such as reverse and gated reverb, phase and stereo imaging, noise reduction and restoration, morphing, synthesis, and other interesting effects.

Figure 5 shows the 26th-order, low-pass filter's impulse response applied by convolution to a test triangle signal. The impulse response graph is obtained by plotting the filter coefficients values. The lack of hard edges in the convolved triangle signal accounts for the loss of high-frequency content.

Finally, to implement a different filter design, simply modify `FIR_TAP_NUM` and the filter coefficients array. Any FIR filter type, beyond low pass, will work, as long as the appropriate coefficients are provided.

ADDING A CONTROL INTERFACE

Let's now implement a simple control interface for the low-pass filter by integrating a push button and a potentiometer. The push button code is based on the `igpio_input_interrupt` example from the MIMXRT1010 SDK, whereas the potentiometer code references the `adc_12b1msps_sar_interrupt` example. Adding both controls primarily involved incorporating relevant includes, global constants and variables, auxiliary functions, and the initialization and peripheral-reading code.

Listing 3 presents code excerpts for the push-button and potentiometer. We utilize the MIMXRT1010's onboard User Button and an external potentiometer connected to input pin A3 on the board. **Figure 6** shows the connection diagram. Lines 4–8 in Listing 3 declare global variables, the purpose of which will become clear shortly.

Lines 12–21 define the interrupt request handlers for the `GPIO` (push button) and `ADC` (potentiometer). Each time the push button is pressed, an interrupt triggers the execution of `EXAMPLE_GPIO_IRQHandler()`. Within this handler, line 13 clears the GPIO interrupt flags, and line 14 sets the `g_InputSignal` variable to true. Line 15 invokes the `SK_ISR_EXIT_BARRIER` macro as a precaution to ensure proper interrupt exit, particularly in cases of

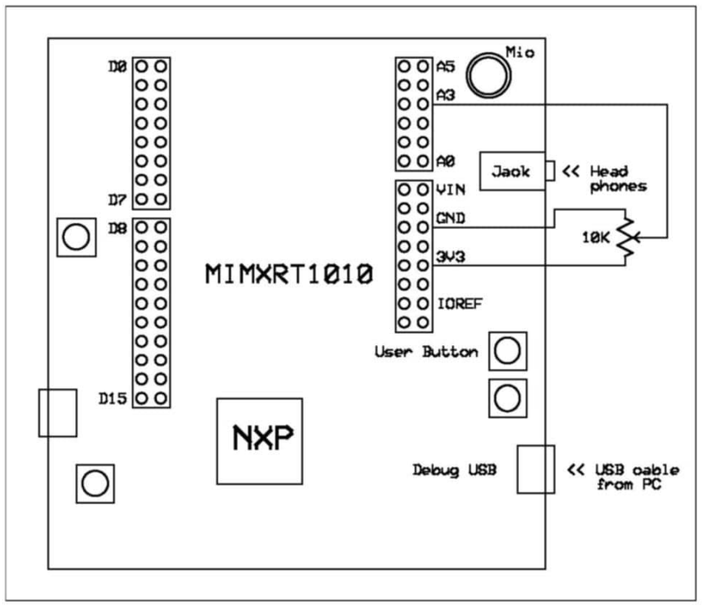


FIGURE 6
Interface connection diagram.

ABOUT THE AUTHOR

Raul Alvarez-Torrico has a BEng in electronics and is the founder of TecBolivia, a company offering services in physical computing and educational robotics in Bolivia. In his spare time, he likes to experiment with wireless sensor networks, robotics, and artificial intelligence. He is also committed to publishing articles and video tutorials about embedded systems and programming in his native language (Spanish), at their company's site www.TecBolivia.com. You may contact him at raul@tecbolivia.com.

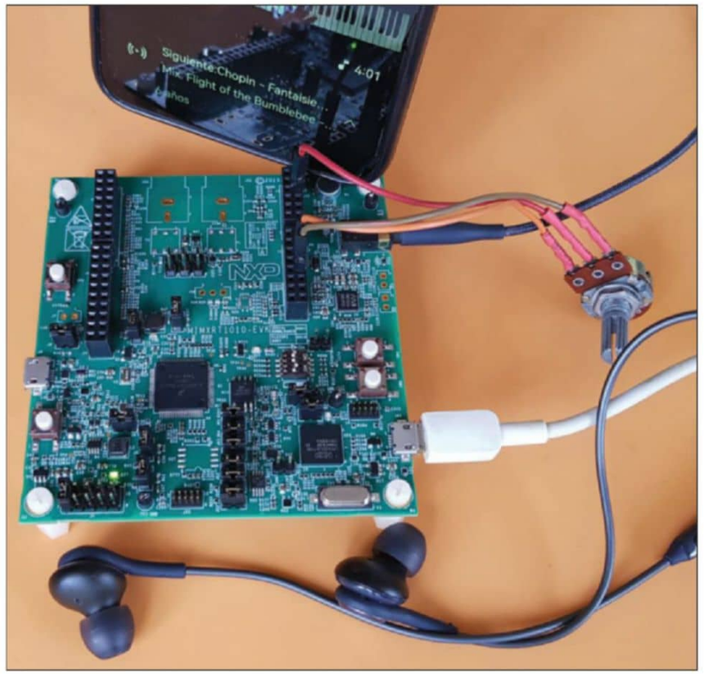


FIGURE 7
MIMXRT1010 board with control interface.

peripheral write delays. This step is critical for complex architectures like the Cortex-M7 core, where calling this macro before exiting any interrupt handler is recommended. Lines 17–21 define the ADC interrupt handler for reading the potentiometer value. This handler sets the `g_AdcConversionDoneFlag` to true and stores the analog input value in the `g_AdcConversionValue` variable.

Lines 26–32 inside the main function handle pushbutton debouncing. Electromechanical contacts, like those in push buttons, often bounce when pressed, causing rapid on-off oscillations lasting approximately 20–30ms. The debounce routine filters these bounces. Line 26 checks if the push button was pressed. If so, line 27 introduces a brief blocking delay to wait out the bouncing period.

Then, line 28 re-reads the push button `GPIO` input to confirm a stable high state. If confirmed, the `g_FilterOn` flag toggles from its default true to false, disabling the low-pass filter. The next button press will re-enable the filter. Line 31 resets the `g_InputSignal` interrupt flag, returning the button to its "unpressed" state.

Next, line 34 triggers a new ADC conversion to read the current potentiometer value. Each successful ADC conversion invokes the `EXAMPLE_ADC_IRQHandler()` interrupt handler. Line 35 declares a `gain` variable to adjust the audio output gain, ranging from 0.0 (no output) to 1.0 (full output). Line 36 calculates the difference (`delta`) between the current and previous potentiometer values. Line 38 checks if the ADC conversion is complete and if the delta is at least 100 units.

If both conditions are met, `g_AdcConversionDoneFlag` is cleared for the next conversion, and line 40 calculates the new output gain by dividing the current ADC value by 4,095. Since the RT1010 MCU uses a 12-bit ADC, potentiometer values will range from 0 to 4,095 (2^{12} - 1). To avoid reacting to electrical noise, the gain is updated only when the potentiometer value changes by 100 or more units. Line 41 stores the current ADC value as "previous" for the next iteration.

Finally, lines 44–63 contain filter code nearly identical to Listing 2, but filtering is applied only when the `g_FilterOn` flag, controlled by the push button, is set. Otherwise, the filter is bypassed. Lines 66–69 implement the delay routine for push-button debouncing. **Figure 7** pictures the board connections to test this code. The potentiometer controls the output gain only when the filter is engaged, which is the default state.

A FIR FOUNDATION

In Part 1 of this series the foundation was laid for developing embedded audio DSP applications, using the NXP i.MX RT1010 Evaluation Kit. The focus was on establishing a straightforward workflow to implementing FIR filters.

By building upon the SDK's audio record-playback example, the C programming language was used to process audio in real-time, leveraging the board's WM8960 codec and hardware peripherals. We also integrated a button and potentiometer as an interface to toggle the FIR filter and adjust gain.

Looking ahead, Part 2 of this series will expand on this foundation by introducing time-delay-based effects, modulation, and distortion, while further refining the control interface for real-time parameter adjustments.

These advancements will enable more complex audio processing tasks, by combining the MIMXRT1010's powerful hardware with accessible filter design software tools. An exciting area for experimenting further with embedded audio DSP lies ahead!

Additional materials from the author are available at:
www.circuitcellar.com/article-materials

References [1] to [2] as marked in the article can be found there.

RESOURCES

NXP | www.nxp.com

embeddedworld
Exhibition&Conference
CONNECTING THE
EMBEDDED COMMUNITY
10 – 12.3.2026
NUREMBERG, GERMANY
JUNITED
Redeem your
ticket voucher
GG4ew26 now!
embedded-world.de/en/codes
Media partners
elektroniknet.de
Markt&Technik
Elektronik
Elektronik automotive
Elektronik medical
NÜRNBERG MESSE

Designing Analog Electronics

Part 2: Signal Conditioning with Op Amps

By
Andrew Levido

Designing analog electronics hardware is less orderly and predictable than digital electronics design. In Part 2 of this article series, Andrew looks at how to design a circuit to extract a small DC signal that happens to be riding on a much larger DC offset.

In Part 1 of this article series (*Circuit Cellar* #424, November, 2025) [1] I introduced the importance of managing error and uncertainty in designing analog circuits. In an example, we conditioned a 12V signal for digitisation by a microcontroller's ADC, and deliberately introduced a small offset to this signal to keep it away from the supply rails to avoid errors.

In Part 2, I will look at the common problem of extracting a small DC signal riding on a much larger DC offset. One example of this might be the measurement of current flowing into or out of the positive terminal of a battery as it charges and discharges (**Figure 1**). We will assume that the battery voltage is a maximum of 24V and we are trying to measure a current up to ±6A.

Current will be measured by inserting a low-value series resistor and monitoring the voltage developed across it. We want to keep the value of the shunt resistor low, since we don't want to drop too much of our precious battery voltage across it, and we want to keep its power dissipation low. We could, for example, use a 25mΩ shunt, which would give us a maximum voltage drop of ±150mV (a 300mV span) and a maximum power dissipation of 900mW. We will therefore need to amplify the shunt voltage by a factor of 10 to get this signal up to a 3V span to suit the microcontroller's analog-to-digital converter (ADC).

The problem is that both ends of the sense resistor are floating at the battery voltage of 24V. The unwanted voltage common to both terminals is much higher than the voltage difference between them that interests us. In formal "electronics-speak," we say that the common-mode voltage on the signal conditioning block's input terminals is 24V, and the differential-mode voltage is ±150mV.

We therefore need a signal conditioning circuit that both amplifies the differential voltage and attenuates the common-mode voltage. This will obviously be quite a bit more complex than the unity gain buffer we used in the example in Part 1, so we will work our way up to it in stages--and cover some useful op amp circuits along the way.

NON-INVERTING AMPLIFIER

The unity-gain buffer is actually a special case of the familiar non-inverting amplifier shown at the top of **Figure 2**. The output of the op amp is fed back to the inverting input via a voltage divider formed by *Rf* and *R1*. We get a unity gain buffer if we set *Rf* to zero.

Negative feedback works to drive the voltage difference between the op amp's two inputs to zero, so the voltage *V'* will be equal to the input voltage V_i, which in turn is equal to the output voltage V_O attenuated by the divider. Rearranging the equation gives an expression for the output voltage in terms of the input voltage. The gain of this circuit is always positive, with a magnitude of 1 or greater. I suppose this circuit is familiar to most readers, but let's take a moment to look a little deeper.

The input impedance of the non-inverting amplifier is equal to the op amp's non-inverting input impedance, so it can be extremely high—especially if a field effect transistor (FET) input op amp is used. The input voltage is limited to the op amp's common-mode input voltage range, which may not extend fully to the supply voltage.

The voltage error at the non-inverting input due to the op amp's bias current (I_B) is entirely dependent on the source impedance R_S. If the input impedance is high, this can be significant. This error, plus the op amp's input offset voltage error (V_{OS}), appear at the op amp's output after being amplified by the circuit gain. The total error voltage at the op amp output is therefore $E_O = G(V_{OS} + I_B R_S)$.

While we are on the topic of errors, the non-inverting amplifier will also have a gain error and associated temperature drift due to the resistor tolerances. This error varies with the ratio of resistances, from zero when configured as a unity-gain buffer, up to the sum of the two resistors' relative errors when *Rf* is much larger than *R1*.

The non-inverting amplifier at the top of Figure 2 works for both DC and AC signals, but if you are only interested in AC signals you may choose to capacitively couple the input as shown at the bottom of the figure. If you do, you must include *R2* to provide a path to ground for the op amp's input bias current. If you leave it out, the bias current will progressively charge *C2*, leading to a large voltage error.

C2 and *R2* will form a high-pass filter with a corner (–3dB) frequency of *1/(2πR2C2)*, so you need to choose the values to suit the bandwidth you require. The circuit input impedance (in the filter's passband) is *R2*, and the error voltage at the input due to bias current is $I_B R2$.

Given that the signal is AC-only, you can also include capacitor *C1* to roll off the gain below the band of interest. The –3dB frequency of this roll-off occurs when the impedance of *C1* is equal to *R1*, and the gain rolls off to unity at DC. This can be handy if your circuit has very high gain, because the op amp's DC input offset and bias error voltages will be subject to a gain of only 1.

The non-inverting amplifier is a useful building block because of its high input impedance, but it is not a flexible circuit. The real power of op amps comes from the next topology, which starts with the inverting amplifier (**Figure 3**, top).

INVERTING AMPLIFIER

The op amp's non-inverting input is fixed to0V, so the negative feedback will force the inverting input voltage to zero too, forming

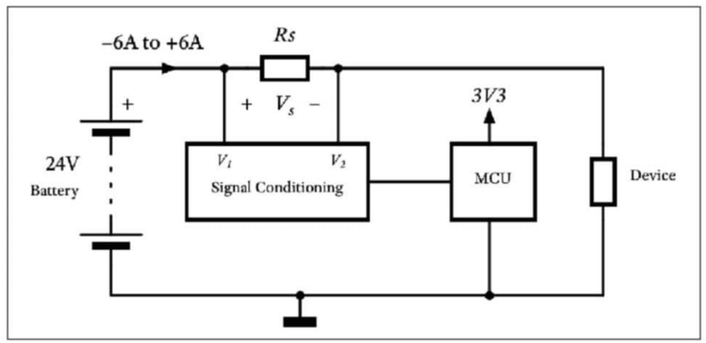


FIGURE 1
The signal-conditioning circuit has to reject the 24V common-mode signal and amplify the differential-mode signal developed across the shunt resistor.

a so-called "virtual ground" that I have indicated with a zero. Ohm's law means the current through *R1* will be V_i / *R1* and that through *Rf* will be V_o / *Rf*. Because (practically) no current flows in or out of the inverting input, these two currents must be equal and opposite, leading to the expression for gain $V_O = -V_i$ *(Rf/R1)*.

The gain is the simple ratio of two resistors (*Rf / R1*), so its magnitude can be greater or less than unity, and the inverting amplifier can provide either gain or attenuation. The input impedance of the circuit is *R1*, not nearly as high as for the non-inverting amp.

Because the gain is a simple ratio between independent resistors, the gain error due to resistor tolerance is the sum of the relative

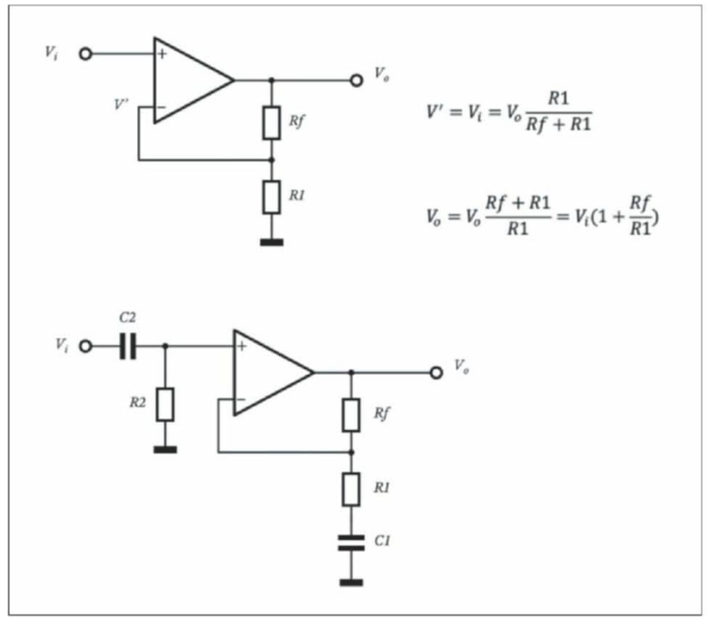


FIGURE 2
The non-inverting amplifier (top) has very high input impedance. A unity-gain buffer is a special case of this circuit, where *Rf* is set to zero.

errors regardless of their ratio. You can probably also see that the gain of this circuit is highly dependent on the source impedance, which is effectively in series with *R1*. Any source appreciable resistance, temperature drift or non-linearity will appear as gain error. For high gain precision, the source impedance should be many orders of magnitude lower than *R1*.

On a more positive note, the common-mode voltage seen by the op amp is always zero, so there is no restriction on the range of input voltage. It can be far outside the supply rails if you choose the resistor values appropriately. The input voltage error due to bias current is proportional *R1* in parallel with *R2* (assuming source resistance is much lower than *R1*), so the voltage error at the op amp's output will be $E_O = G(V_{OS} + I_B\, R1||Rf)$.

We can reduce the bias current error by adding a third resistor to this circuit (middle of Figure 3). The resistance seen by both inputs is now the same, so the error voltage due to bias current is cancelled out. We are

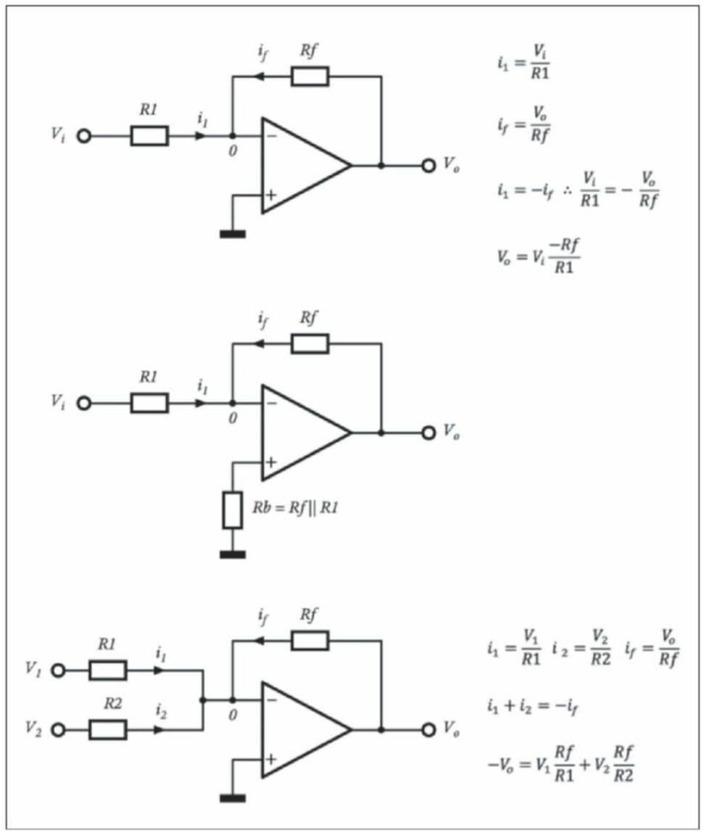


FIGURE 3
The inverting amplifier is a particularly flexible circuit, and is the basis of many op amp circuits, including the summing amplifier shown at the bottom. The disadvantage of this circuit is low input impedance and the effect of source resistance on gain.

therefore left with just the error due to the input offset current (I_{OS}), which you will recall from Part 1 is somewhat lower than I_B. The error at the output of this circuit will therefore be $E_O = G(V_{OS} + I_{OS}\, Rb)$.

At the bottom of Figure 3 is an inverting amplifier with two inputs, often called a "summing amplifier." This time the feedback current is equal in magnitude (but opposite in sign) to the sum of the two input currents. Since the input and output voltages are proportional to their respective currents, the circuit effectively adds the two input voltages. *R1* and *R2* can be chosen to weight each input independently, and *Rf* can be used to scale the overall result.

There are literally hundreds more interesting and useful op amp circuits we could look into, but now it is time to return to our design problem. If you refer back to Figure 1 you can see that we would like our conditioning circuit to produce an output proportional to $+V_1 - V_2$. We really want a circuit with one inverting input and one non-inverting input.

DIFFERENCE AMPLIFIER

There is such a thing as a "difference amplifier," and it is shown in **Figure 4**. It is sometimes called a "differential amplifier," and as its name suggests, it amplifies the difference between the input voltages. To analyze it, we can use a handy technique known as "superposition." Superposition means that in any linear circuit, the overall effect of multiple inputs is given by the sum of their individual effects. In this example, we can determine the output of the circuit due to input V_1 while V_2 is grounded (red) and the output due to V_2 with V_1 grounded (green), and then add the two expressions to get the output due to both.

With V_2 grounded, as far as V_1 is concerned, the circuit is a non-inverting amplifier preceded by a voltage divider, so $V_{O(1)}$ is equal to V_1 *(R2/R1)*. When V_1 is grounded, the circuit is an inverting amplifier for V_2, so the output $V_{O(2)}$ is equal to $-V_2$ *(R2/R1)*. Adding the two gives us the result shown at the bottom of the figure; $V_O = (V_1 - V_2)(R2/R1)$.

This result means that any voltage common to both input terminals does not appear at the output. but any voltage difference between two inputs is amplified by a factor of *R2/R1*. More formally we say the differential-mode gain is *R2/R1* and the common-mode gain is zero. The common-mode gain will truly be zero only if the resistor pairs are perfectly matched--and we already know that in real circuits this won't be the case. There must therefore be some (hopefully very small) common-mode gain.

COMMON-MODE REJECTION RATIO (CMRR)

Rather than talk about common-mode gain, in this context we typically use the "common-mode rejection ratio" or CMRR. This is the ratio between the differential-mode gain and the common-mode gain, expressed as a logarithm of powers. Don't be intimidated by that last sentence, since all you need to know is that CMRR = $20 \log(G_{DM}/G_{CM})$. For example, a difference amplifier with common-mode gain of 1/100 and differential-mode gain of 10 would have a CMRR of 60dB.

If we were to build a difference amplifier for our application, with a differential-mode gain of 10, using 1% tolerance resistors we could expect a worst-case CMRR of 54dB, meaning the common mode voltage would be attenuated by a factor of 500 relative to the differential-mode voltage This is not a lot. In our example with a common-mode voltage of 24V, the difference amplifier would have around 48mV residual error on its output. Given we have ±1.5V full scale at the output of the signal conditioning block, we are looking at 3.2% error. To get reasonable levels

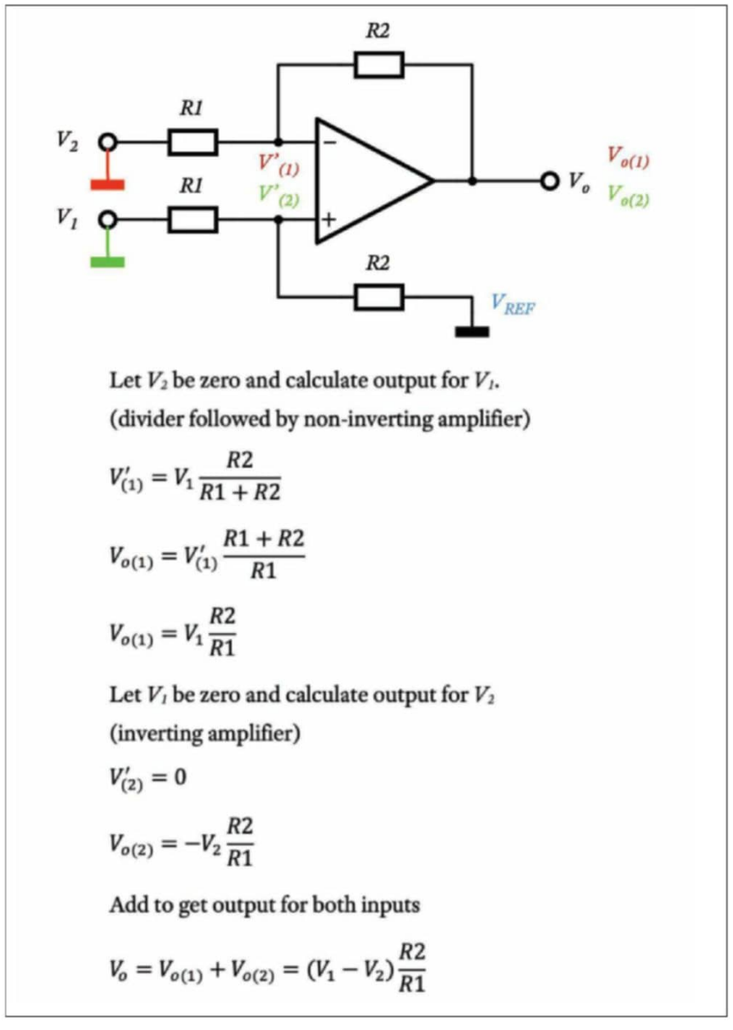


FIGURE 4
You can use "superposition" to determine the transfer function of this difference amplifier. You calculate the gain of each input in turn (with the others grounded), and add the resulting expressions.

LCR-Reader®MPA/R2/R3
Professional Multimeters
Unique product line with:
Basic accuracy of 0.1%
Up to 250 kHz
Unrivaled Ranges:
0.1 pF to 1 F
1 nH to 100 H
5 mΩ to 20 MΩ
Choice of models with different functionalities
2020 Test Engineering PRODUCT of the YEAR Bronze Award
Optional Bluetooth model for data logging and quick Pass/Fail component assessment
LCR/ESR
Frequency
Oscilloscope
AC/DC Voltage
Signal Generator
AC/DC Current
LCR-Reader®
Budget LCR Meter
LCR/ESR measurements
1 oz. weight
Automatic range selection
One button navigation
OLED display
Basic Accuracy: 0.5%
Test Frequency: 100 Hz, 1, 10 kHz
Test Signal Level: 0.5 +/- 5% Vrms
Made in Canada
Siborg Systems Inc.
24 Combermere Crescent, Waterloo, Ontario, Canada N2L 5B1
Phone: 1-519-888-9906 Fax: 1-519-725-9522 www.Siborg.com
www.LCR-Reader.com
SIBORG SYSTEMS INC

of CMRR out of this circuit, you need to use precision-matched resistors.

Fortunately you don't have to roll your own if you don't want to. You can buy difference amplifiers with precision-matched resistors integrated onto the chip for reasonable prices. There is a huge range available, and they can generally achieve CMRRs in the 80dB to 120dB range. The LT1997-1 from Analog Devices is a particularly nice example. It includes resistors matched to a precision of 0.002%, which give it remarkable performance.

Figure 5 is a block diagram of what's inside. It shows three different sets of input resistors for gains of 10, 20, or 50. You can connect these inputs in various combinations to get other gains, too. With a gain of 10 that our application requires, the specs are impressive. CMRR is a minimum of 106dB at 25°C, and 104dB over the full −40°C to 125°C operating temperature range. Worst-case gain error is 0.012% with ±1ppm/°C drift. Input offset voltage is ±65µV with 1.5µV/°C drift. The outputs will swing to within 50mV of either rail at light loads. Supply voltage can be anywhere between 3.0V and 50V-- a spec that will come in handy later.

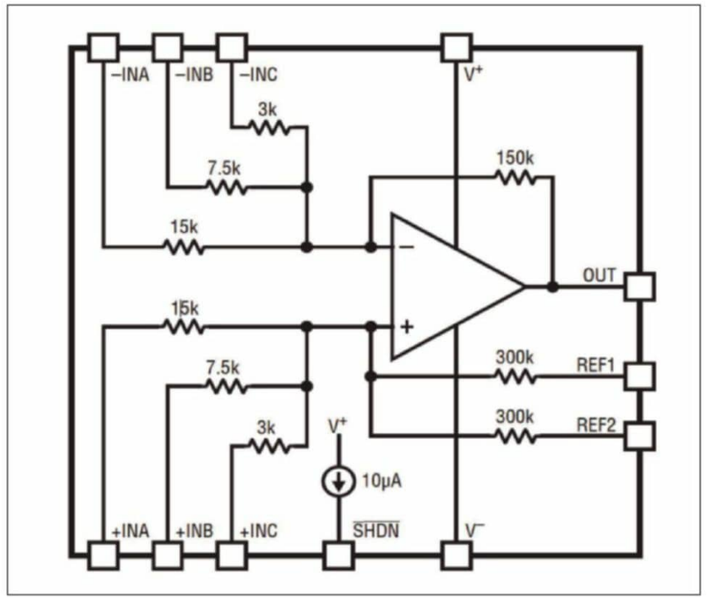


FIGURE 5
This is the circuit of the LT1997-1 differential amp extracted from the datasheet [2]. Three pairs of input resistances offer different gains. See the text for an explanation of the two 300kΩ resistors at the bottom right.

REFERENCE INPUT

If you look closely at Figure 5 you will notice the two 300kΩ resistors connected to the REF1 and REF2 pins. If both of these pins are connected to ground, you get the same circuit as the difference amp shown in Figure 4. However, the designers have left them open, so we can add an offset to the difference amplifier's output.

If the ground end of the lower *R2* in Figure 4 were connected to a third voltage V_{REF}, you could add a third term to the superposition result by assuming the V_1 and V_2 inputs were grounded. The expression for output voltage then becomes $V_O = (V_1 - V_2)(R2/R1) + V_{REF}$. The voltage at the V_{REF} input appears as on offset at the output.

This is useful in our application, because without any offset, the ±150mV shunt voltage would appear at the difference amp's output as a ±1.5V signal. We want to shift this up by 1.65V (half of the 3.3V logic supply) to get a signal in the range of 0.15V to 3.15V, which is safely away from the power rails, but using most of the ADC range.

This is where the splitting of the reference resistor within the LT1997-1 comes in. Connecting REF1 to 3.3V and REF2 to ground (or *vice versa*) is the same as connecting a 1.65V source via a 150kΩ resistor. The divider matching error is just 0.006%, so the offset error will be well within the ±0.5 LSB of the ADC's half-scale value. We don't even need to digitize the offset and subtract it in code like we did in Part 1.

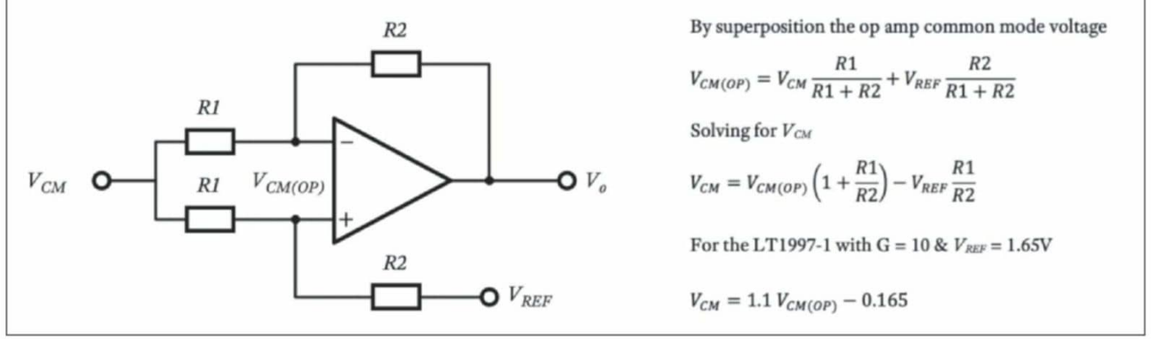


FIGURE 6
We use "superposition" to calculate the permissible range of common mode voltage for the difference amplifier.

COMMON-MODE VOLTAGE

There is still the matter of common-mode voltage to consider. Although the circuit common-mode voltage can fall outside the supply rails, we still have to keep the common-mode voltage at the op amp within its V_{CM} limits In the case of the LT1997-1, this is between V^+ – 1.75V and V^- +1.75V. The permissible input common-mode range is calculated using superposition according to **Figure 6**.

With our component values, the common mode range turns out to be $1.1V_{CM(OP)} - 0.165V$. This suggests that we will need to use a relatively high supply voltage to get the +24V common-mode input we want. For example, if we powered the LT1997-1 from 3.3V, the maximum V_{CM} we could tolerate would be just over 1.5V. If instead we use the 24V battery supply to power the LT1997-1, we get a maximum common-mode voltage of 24.3V and a 1.8V minimum common-mode voltage.

In case you think the 24.3V maximum is a litle close to 24V nominal, consider that if the battery voltage rises, the allowable common-mode voltage will rise as well. If you use a difference amplifier, please check the common-mode voltage limits. I have seen even experienced engineers caught out on this more than once.

The complete signal conditioning circuit, shown in **Figure 7**, looks deceptively simple, but as we have seen there is a lot going on under the hood. The LT1997-1 chip is not cheap at around $10 in small quantities, but it is hard to beat for precision, and it eliminates a lot of external circuitry. In Part 1, I created an error table for the analog part of this circuit, so you could see the various sources of error. I ignored temperature-dependent errors, just to keep things simple. That table is reprinted here for your reference (see **Table 1**).

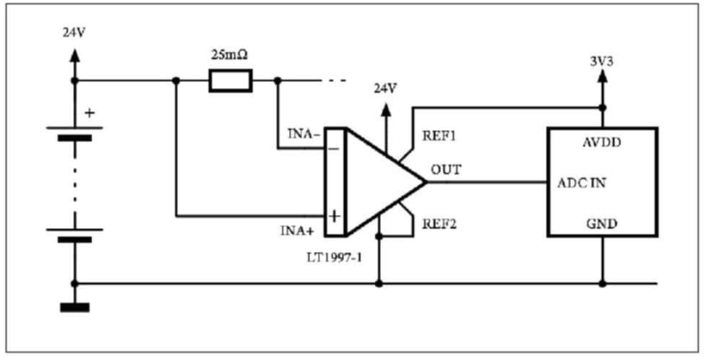


FIGURE 7
Our final signal-conditioning circuit for the problem of Figure 1 is deceptively simple. There is, however, a lot going on under the surface.

The shunt resistor tolerance is far and away the largest source of error contributing 1% of the 1.04% total untrimmed error. You might use a tighter tolerance shunt, but high-precision, low-value power resistors are hard to come by and very expensive if you can find them. You can, of course, trim out much of this error.

I think trimmed errors in the 0.02% to 0.05% range would be realistic, depending on the trimming strategy. This corresponds to 1 part in 2,000, and is therefore consistent with the maximum sort of precision one might get from a microcontroller ADC. Of course, there will be some temperature-

	Errors	At Nominal 25°C		
		Nominal Value	Absolute Error	Relative Error
1	Shunt Resistor (Bournes CRA2512, 25mΩ, 1%)	25.0E-03	250.0E-06	1.000%
2	Shunt Voltage (@ 6A)	150.000E-03	1.50E-03	1.000%
3	Diff Amp Input Offset Error (LT1997-1, 65µV)	000.0E+00	65.0E-06	0.043%
4	Diff Amp Offset Current Error (LT1997-1, 3nA, 0.25Ω) - negligible	000.0E+00	000.0E+00	0.000%
5	Total Untrimmed Voltage Error at Diff Amp Input (Lines 2+3+4)	000.0E+00	1.57E-03	1.043%
6	Diff Amp Gain (LT1997-1 x10 Inputs , 0.012%)	10.00E+00	1.2E-03	0.012%
7	Voltage error due to gain error (Line 6* 1.5V)	000.00E+00	180.0E-06	0.012%
8	Voltage error at output due to input error (Line 5 *gain)	000.00E+00	15.7E-03	1.043%
9	CMRR Error at outputm(24V, G = 10, 106dB)	000.00E+00	1.2E-03	0.080%
10	Zero error due to REF resistor ratio (0.006%)	000.00E+00	99.0E-06	0.006%
11	Total Untrimmed Error(Line 7+ 8 + 9 + 10)	000.00E+00	17.1E-03	1.038%
12	Total Trimmed Error (estimated)	000.00E+00	660.0E-06	0.040%

TABLE 1
I reprinted this table from Part 1 of this article series. This is an error table for the analog part of this circuit, so you can see the various sources of error.

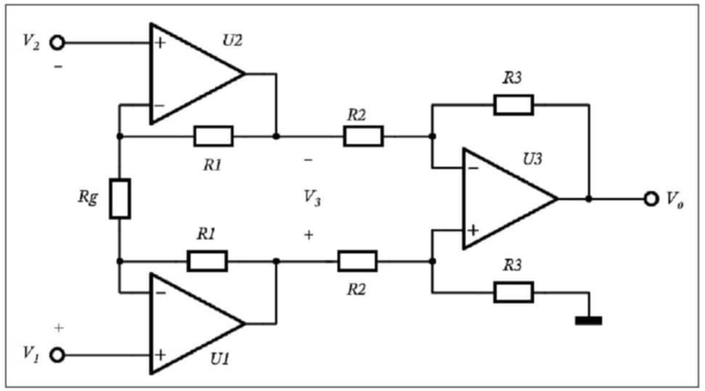


FIGURE 8
A classic, three op amp instrumentation amplifier combines high CMRR with high input impedance. As a bonus, the gain can be set by just one resistor, *Rg*.

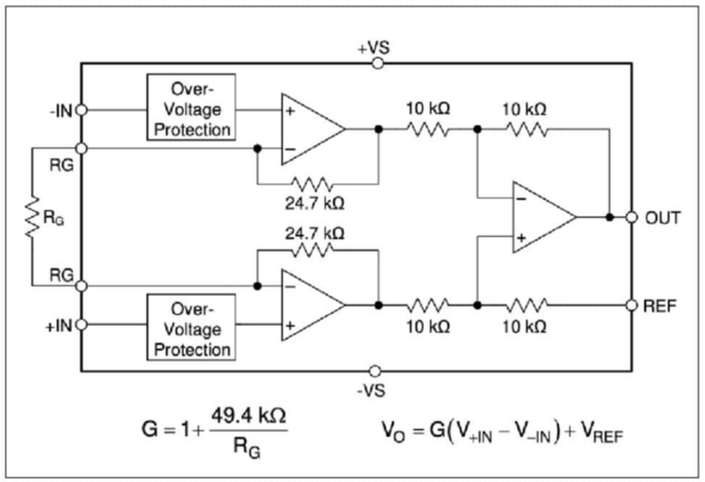


FIGURE 9
The INA821 is a typical example of an off-the-shelf instrumentation amplifier. (Circuit reproduced from the datasheet [3].) The reference output allows the output to be offset by an arbitrary voltage.

ABOUT THE AUTHOR

Andrew Levido, (andrew.levido@gmail.com) earned a Bachelor's degree in electrical engineering in Sydney, Australia. He worked for many years in R&D and management roles for power electronics, telecommunication, and IT companies. Andrew has maintained a hands-on interest in electronics, particularly embedded systems, power electronics, and precision analog design. Over the years, he has written many technical and educational articles for various publications. He also provides consulting services to a number of clients.

Additional materials from the author are available at: www.circuitcellar.com/article-materials

References [1] to [2] as marked in the article can be found there.

dependent errors, which I have conveniently ignored.

INSTRUMENTATION AMPLIFIERS

The input impedance of the difference amplifier is fairly low at 2*R1* for differential-mode signals and 0.5(*R1*+*R2*) for common-mode signals. Like the inverting amplifier, source impedance will affect gain, and in addition, source impedance imbalance will negatively affect the CMRR. This means you can really use only difference amplifiers in circuits with very low source impedance (like ours).

We can overcome these limitations with the circuit shown in **Figure 8**. The difference amp (*U3*) is preceded by a pair of coupled non-inverting gain stages (*U1* and *U2*). This arrangement is known as an instrumentation amplifier. The input stage feedback works to keep the inverting inputs of *U1* and *U2* at V_1 and V_2 respectively, so their difference is maintained across *Rg*. You can then use superposition to calculate the first stage output voltage $V_3 = (V_1 - V_2)[1 + 2(R1/Rg)]$. The gain of the input stage for common-mode signals is 1.

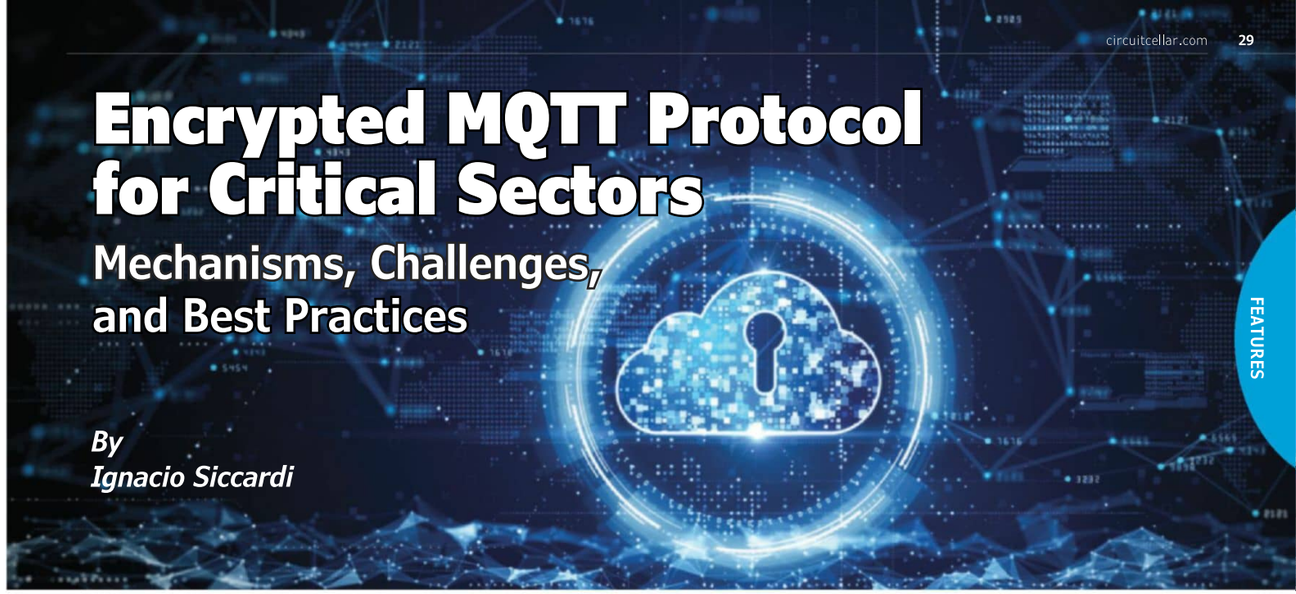
In a typical instrumentation amplifier the difference amplifier stage has a low gain (often unity) but a high CMRR. The gain is provided by the input stage, which has a CMRR equal to gain, since the common-mode gain is always 1. The total CMRR is the sum of the two.

The INA821 from Texas Instruments is a typical example of an instrumentation amplifier. Its internal circuit is shown in **Figure 9**. All the resistors except the single gain-setting resistor *Rg* are included on the die. These are precision matched and provide a minimum CMRR of 92dB at a gain of 1 (when the input stage provides no additional CMRR). If we were to configure this for a gain of 10 (*Rg* = 5.49kΩ), we would get an additional 20dB CMRR, for a total of 112dB.

Both common-mode and differential-mode input impedances are more than 100GΩ, and the input offset voltage is ±35µV with 5µV/°C drift. The main limitation compared to a difference amp is that their common-mode input voltage range does not extend past the supply rails, and often not quite to them. In the case of the INA82, for example, the common-mode input range can't be closer than 2V to either supply rail.

THE JOY OF OP AMPS

Op amps are extremely useful and flexible devices that truly are the basic building blocks of modern analog electronics. We have now had a good look at their use in DC circuits so, in Part 3 of this article series, we will ook at their AC characteristics and explore some useful filter topologies.



Encrypted MQTT Protocol for Critical Sectors

Mechanisms, Challenges, and Best Practices

By
Ignacio Siccardi

MQTT is considered the standard messaging protocol for the Internet of Things. In this article, Ignacio outlines the need for MQTT encryption in critical sectors such as healthcare and financial institutions, encryption mechanisms, best practices, and challenges

The expansion of the Internet of Things (IoT) into industries such as healthcare, energy, finance, and transportation has elevated the importance of secure, reliable communication protocols. Among the most widely adopted protocols for machine-to-machine communication is the Message Queuing Telemetry Transport (MQTT).

MQTT is a publish/subscribe protocol for machine-to-machine messaging. Designed for lightweight and low-bandwidth environments, MQTT has become a cornerstone in connecting IoT devices that range from medical sensors to smart grid controllers. It is considered an OASIS standard messaging protocol for the IoT.

However, as IoT deployments move deeper into critical sectors, the potential risks of cyberattacks, data breaches, and unauthorized access grow significantly.

A compromised IoT system in a hospital, a power plant, or a financial institution can cause not only monetary damage but also physical harm and reputational loss. This is where encryption within MQTT communications plays a pivotal role.

ENCRYPTION IN CRITICAL SECTORS

The original MQTT specification was designed for simplicity, reliability, and efficiency, but not for strong native security. As IoT scaled into mission-critical applications, several concerns emerged:

- *Data confidentiality:* Sensitive data such as patient records, energy consumption patterns, or transaction details can be intercepted if MQTT traffic is left unencrypted.
- *Data integrity:* Attackers could manipulate unprotected MQTT messages, sending false commands to industrial machinery or injecting malicious payloads.
- *Authentication gaps:* MQTT by itself provides only basic username and password authentication, which can be insufficient against sophisticated attackers.
- *Compliance requirements:* Regulations such as HIPAA (healthcare), PCI-DSS (finance), or NERC CIP (energy) demand secure data handling. Unencrypted MQTT traffic would be non-compliant.

Encryption ensures that even if network traffic is intercepted, the data remains unreadable without the appropriate decryption keys. For critical infrastructures, encryption is not optional—it is a regulatory and operational necessity.

ENCRYPTION MECHANISMS FOR MQTT

To address security challenges, encryption can be integrated into MQTT communications primarily through Transport Layer Security (TLS). TLS, the same protocol that secures HTTPS, provides confidentiality, integrity, and authentication.

TLS/SSL Over TCP

- *How it works*: The MQTT broker and clients establish a TLS handshake, creating an encrypted channel before message exchange, as shown in **Figure 1**.

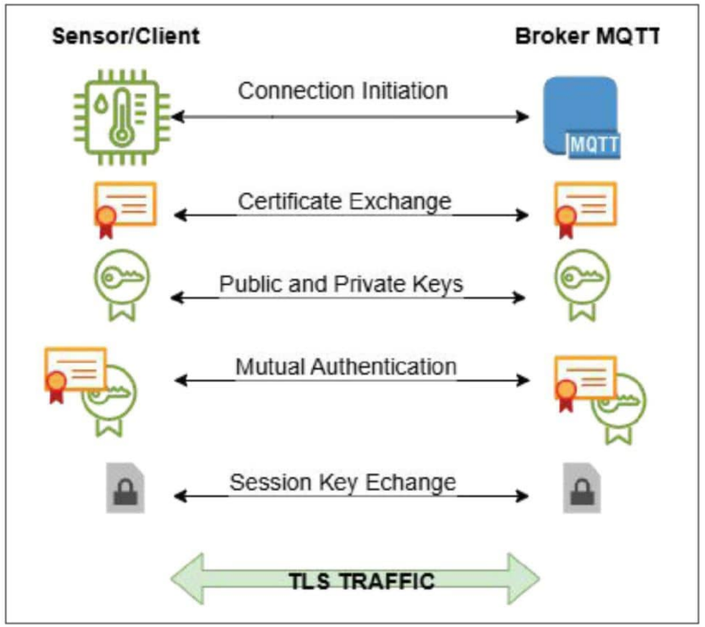


FIGURE 1:
An encrypted tunnel allows secure MQTT data exchange for sensitive information.

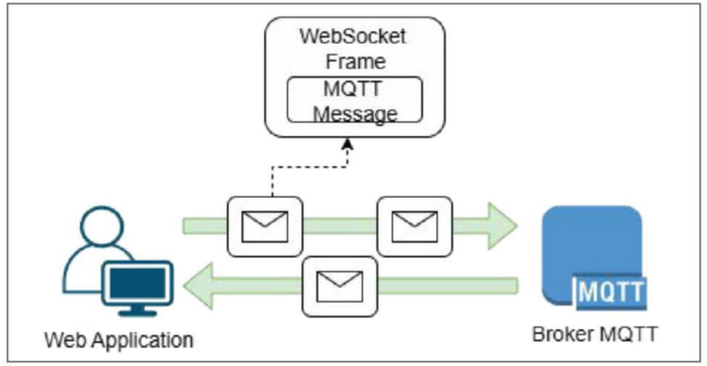


FIGURE 2:
MQTT encrypted through WebSockets provides security within the technology stack of web sites.

ABOUT THE AUTHOR

Ignacio Siccardi is an Information Systems Engineer specializing in IT infrastructure and IoT. He currently works on the deployment of IoT infrastructure with LoRaWAN in Argentine agro-industries.
You can learn more about Ignacio at his personal blog www.rosariot.com.

Additional materials from the author are available at:
www.circuitcellar.com/article-materials

RESOURCES

EMQX | www.emqx.com

- *Advantages*: Strong, industry-standard security; support for mutual authentication using client and server certificates.
- *Use cases:* Healthcare devices transmitting patient vitals, financial systems sending trading updates.

MQTT Over Websockets with TLS

- *How it works:* MQTT messages are encapsulated within WebSocket frames and secured with TLS as illustrated in **Figure 2**.
- *Advantages*: Useful for web-based IoT dashboards and remote monitoring. While more complex than TLS over TCP, this method has much to recommend it when human factors and a web interface are part of the system design.
- *Use cases*: Control centers in energy and transportation sectors where operators use browser-based tools.

End-to-End Application-Level Encryption

- *How it works*: Data payloads are encrypted by the sender before being published, and only the intended subscribers can decrypt them, illustrated in **Figure 3.**
- *Advantages*: Protects data even if the broker is compromised.
- *Use cases*: Defense and military applications where absolute secrecy is required.

BEST PRACTICES FOR DEPLOYING ENCRYPTED MQTT

Organizations in critical sectors must treat security as a continuous process rather than a one-time configuration. The following practices are widely recognized as essential:

- *Enforce TLS 1.2 or higher:* Older versions of SSL and TLS have known vulnerabilities. Using TLS 1.2 or 1.3 ensures stronger encryption and reduced exposure.
- *Use strong authentication mechanisms*: Beyond simple username/password, consider mutual TLS (mTLS) with digital certificates to authenticate both broker and clients.
- *Implement fine-grained authorization*: Access control lists (ACLs) should define which topics each client can publish or subscribe to. This prevents unauthorized access or data leakage.
- *Rotate keys and certificates regularly:* Regular renewal and revocation of certificates reduce the risk of compromised credentials being exploited.
- *Monitor and audit MQTT traffic:* Security monitoring tools can detect anomalies such as unusual publishing frequency, unexpected topics, or failed authentication attempts.
- *Segment IoT networks*: Critical MQTT traffic should run on isolated networks or virtual LANs (VLANs) to minimize exposure.

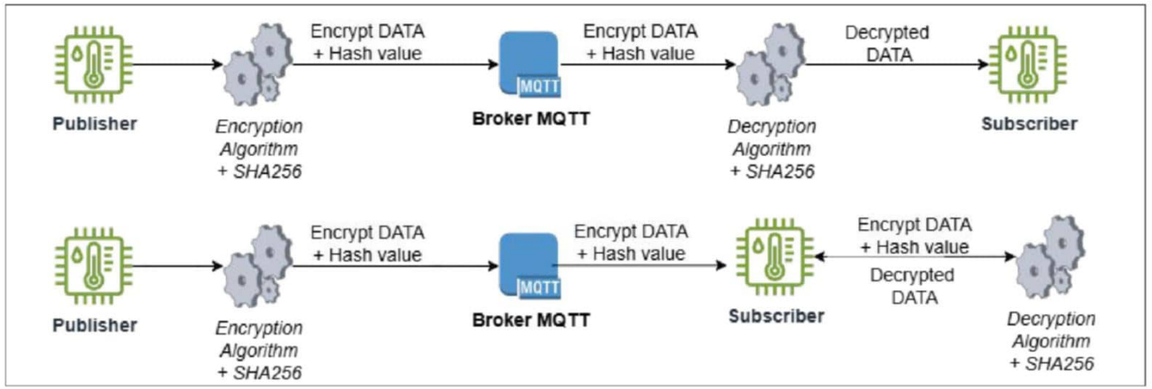
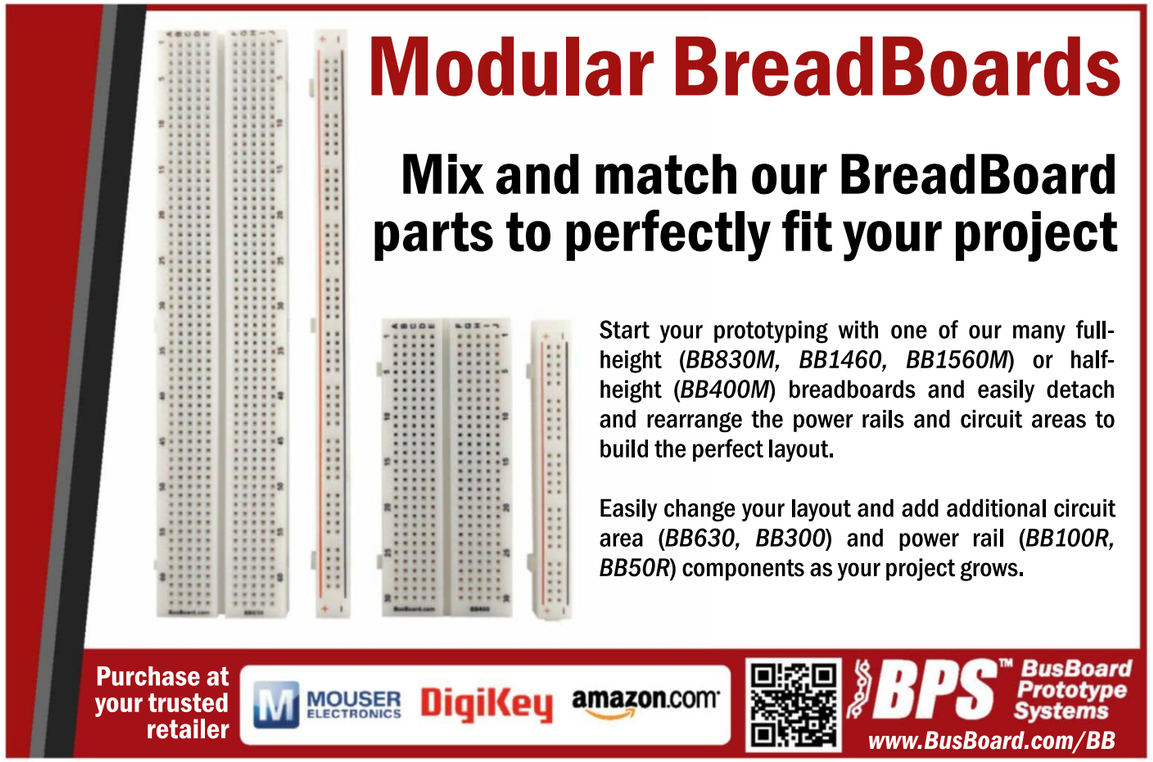


FIGURE 3:
Application-level encryption is ideal for secure communications between applications or individuals trusted by one another.

CHALLENGES IN IMPLEMENTING ENCRYPTED MQTT

Although encryption enhances security, its implementation in resource-constrained environments is not without challenges. Some of these challenges are noted below.

- Processing overhead: TLS encryption and decryption require CPU and memory resources. Low-power IoT devices may struggle to handle strong cryptographic operations.
- Latency considerations: Handshakes and certificate validations can add milliseconds of delay, which may affect real-time control systems.
- Certificate management: Large IoT deployments with thousands of devices face difficulties in managing digital certificates efficiently.
- Backward compatibility: Legacy devices may not support TLS, forcing operators to choose between upgrading hardware or introducing secure protocols via external software components.


Modular BreadBoards
Mix and match our BreadBoard parts to perfectly fit your project
Start your prototyping with one of our many full-height (BB830M, BB1460, BB1560M) or half-height (BB400M) breadboards and easily detach and rearrange the power rails and circuit areas to build the perfect layout.
Easily change your layout and add additional circuit area (BB630, BB300) and power rail (BB100R, BB50R) components as your project grows.
Purchase at your trusted retailer
MOUSER ELECTRONICS
DigiKey
amazon.com
BPS BusBoard Prototype Systems
www.BusBoard.com/BB

TECH FEATURE

Sensors in the Spotlight

The Next Decade of Embedded Sensor Systems

By
Curtis Franklin,
Editor-In-Chief

Sensors, in all their types and functions, are the means by which the real world is introduced to an embedded system. No matter what data the system needs about the physical environment, sensors are how that data is collected and provided to the processor.

Perhaps because sensors are so common and necessary, it can be easy to lose sight of the fact that they are not frozen in technological time. While different types of sensors evolve at varying rates, it's rare to find a modern sensor that hasn't developed from its predecessors of even a few years ago.

Embedded sensors have rapidly transitioned from simple transducers to deeply intelligent, networked participants in the modern cyber-physical world. For engineers on the frontline of systems design, understanding not just the immediate landscape—but the forces rapidly shaping its future—is paramount. By 2030, embedded sensor networks will form the bedrock of automation, intelligence at the edge, sustainability, security, and new digital services.

A single article on the broad topic of "sensors" can be somewhat daunting, as the topic encompasses a wide range of functions and technologies. However, that same breadth makes the topic one that's both vital and exciting—sensors are, in many ways, the foundation on which embedded systems are built, as they provide the data on which critical control decisions are made.

In most cases, sensors are built to deliver a single type of data to the system. That conceptual simplicity doesn't mean they're exempt from the influence of AI, which has been a prime driver of much of the computing world in this decade.

To better understand where sensors are today and where they're headed in the next few years, it's essential to examine the technology, market drivers, and breakthrough innovations that will shape the next decade in sensor-enabled embedded systems. Whether designing solutions for the automotive, industrial automation, healthcare, or emerging smart infrastructure sectors, it's important to understand sensor architectures, integration practices, and future-proofing strategies.

SENSOR FUNDAMENTALS

Sensors convert physical phenomena into electrical signals that are readable by embedded systems, enabling their interaction with the real world. Because there are sensors for various physical phenomena, a wide range of technologies and methods are employed within different sensors. There are several key sensor categories.

Temperature sensors, including thermistors, thermocouples, resistance temperature detectors (RTDs), and digital sensors, play a crucial role in facilitating thermal management, process control, safe worker condition compliance, and environmental monitoring.

Temperature sensors such as the Dwyer S2-1C Surface Mount Temperature Sensor 3K Thermistor [1] (shown in **Figure 1**) detect changes in temperature by exploiting physical properties that vary predictably with heat. These properties include electrical resistance (used in RTDs and thermistors); voltage generation (used in thermocouples); and infrared radiation (used in non-contact IR sensors).

Today, miniaturized IC-based sensors (e.g., DS18B20) dominate consumer and industrial applications.

Proximity and distance sensors, including ultrasonic, IR, and LiDAR sensors, utilize various energy types to determine the distance from the sensor, supporting applications such as robotics, mobility, gesture recognition, and automation. These sensors work by emitting bursts of energy in either the sound or light parts of the spectrum and measuring the echo time to calculate the distance between the sensor and an object.

Motion and orientation sensors measure acceleration forces using microelectromechanical systems (MEMS), including accelerometers, gyroscopes, magnetometers, and inertial measurement units (IMUs). They provide data on the device's orientation and motion in space, enabling applications such as robotics, drones, mobile devices, and gaming.

Pressure sensors, such as barometric, piezoelectric, and capacitive designs, serve various applications across the automotive, industrial, medical, and meteorological fields. They change resistance based on physical input. That change is used in voltage divider circuits to produce

Light sensors utilize light-emitting and light-detecting components to measure the reflection or intensity of different parts of the light spectrum, ranging from ultraviolet to infrared and the visible spectrum in between. These sensors, which include photodiodes, light-dependent resistors (LDRs), and ambient light ICs, enable smart lighting, display adjustment, and environmental analytics in home, business, and industrial automation.

Gas and air quality sensors, including electrochemical, metal oxide, and optical devices, work by detecting specific pollutants—such as particulate matter, volatile organic compounds (VOCs), and gases like CO_2 or CO—using technologies that include laser scattering and electrochemical reactions. These sensors convert environmental data into electrical signals that can be interpreted in real time—vital in smart building, industrial safety, and environmental control applications.

Biometric, touch, and capacitive sensors are utilized for authentication, health monitoring, and user interface applications. A variety of sensors are used for biometric and touch input purposes.

For example, fingerprint scanners use capacitive or optical imaging to capture ridge patterns while iris scanners capture detailed patterns in the colored ring around the pupil. Facial recognition scanners utilize 2D/3D imaging and feature mapping (such as the eyes, nose, and jawline), while voice recognition sensors analyze the pitch, tone, and cadence of speech to identify unique voice characteristics for user authentication and voice command applications.

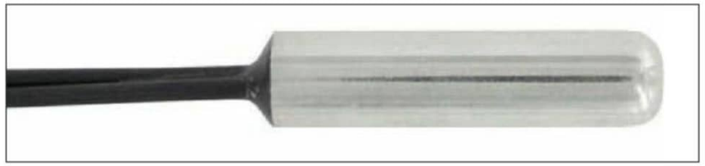


FIGURE 1
Surface-mount temperature sensors are space-sensitive options for designers.

Other sensors include humidity, magnetic field, sound, vibrational, color, position, and networked RFID/NFC sensors, among others, to complete the array of sensors necessary to provide a complete picture of the environment to an application.

INTEGRATING SENSORS

Today's embedded systems depend heavily on both analog and digital sensors. Fortunately, the technology exists to make integrating those sensors into a solution considerably easier than it was even a decade ago.

The first level of integration involves analog sensors. In many cases these will require analog-to-digital converters (ADCs) and precision signal conditioning as part of the hardware chain in order to function.

Digital sensors simplify integration, leveraging protocols such as I^2C, SPI, UART, CAN, and modern wireless standards (BLE, LoRaWAN, Zigbee, Wi-Fi 6/7) for communication between sensors and microcontrollers, as well as among mesh architectures of sensors. It's possible, in many cases, to use the same protocols with analog sensors after an ADC is deployed; however, digital sensors often provide a simpler integration route.

Advanced designs increasingly feature sensor fusion, where multiple sensor modalities combine for more reliable, context-rich data and smarter system responses. Examples abound in navigation (vehicle Advanced Driver-Assistance Systems (ADAS)), environmental analytics (smart cities), and health wearables.

SENSOR MARKET FORCES

Sensors have become indispensable in multiple sectors, each contributing a significant share to the $15 billion sensor market, projected to reach over $40 billion by 2033.

The various market segments may require different sensors, but each depends on a combination of sensors, interfaces,

2165451

TECH FEATURE

FIGURE 2
LiDAR sensors are changing fields ranging from self-driving cars to archaeology.

and controllers to properly respond to the environmental and application conditions it is presented with.

Automotive and Transportation—Sensors are the backbone of ADAS, autonomous mobility, safety instrumentation, EV power management, and infotainment. MEMS, LiDAR, including the SICK unit [2] shown in **Figure 2**, radar, and environmental sensors support features from lane keeping and collision avoidance to fleet optimization. Regulatory and safety demands, as well as electric vehicle trends, are accelerating sensor deployment.

Most new vehicles carry dozens, if not scores, of sensors for the various purposes listed above. And the cost of the sensors can figure significantly in the overall cost of the vehicle. A single LIDAR sensor, for example, can have a cost ranging from as low as $10 to as high as $20,000, depending on integration, range, and accuracy.

As more companies move into cars that have ADAS and self-driving features, the importance (and number) of sensors will grow. As the sophistication of the sensor collection grows, it is pushing the sophistication of the networks that connect those sensors into an application delivery system.

Healthcare and Medical Devices—The miniature submarine of "Fantastic Voyage" has not yet become part of the medical tool chest, but a wide array of sensor-based diagnostic tools is now in regular use. Miniaturized, biocompatible, and networked sensors are propelling personalized medicine, real-time remote monitoring, tele-health, and wearable analytics to detect conditions and aid in the diagnostics of health concerns that range from the commonplace to the rare.

Pressure, motion, heart rate, arterial oxygen saturation (SpO2), and glucose-monitoring sensors are among those that form the foundation of products shifting diagnostics and chronic care to patient-centric, continuous platforms.

While every sensor used in a product must conform to the standards and specifications of the regulations that govern the industry, medical devices are understandably subject to considerably more rigorous scrutiny than most. Among other things, the standards for components used in medical devices tend to require higher precision and lower tolerance drift than those for many other industries.

Industrial Automation and Manufacturing—The smart factory revolution—powered by Industry 4.0 [3]—relies on sensors for predictive maintenance, machine vision, process control, robotics, and efficient resource monitoring. While much of the conversation around Industry 4.0 has centered on software—computer resource management (CRM) beginning in the 1990s and AI in the 2020s—neither would be able to do their respective jobs without data from a fleet of sensors deployed at every point of the manufacturing process.

Sensor-driven applications enable automation, providing reduced downtime, improved quality assurance, and more agile, adaptive manufacturing. The sensors deployed in industrial applications must function in a wide range of environmental conditions, from temperature extremes to excessive vibration, as well as dust and moisture conditions that test the limits of both the sensors and their surrounding systems.

In addition, sensor networking can pose a challenge due to the RF environment in which many industrial systems operate. Sensor interfaces are critical elements for creating sensor infrastructures that can overcome RF saturation limitations; however, these interfaces inevitably add cost and complexity to the overall sensor packages.

Consumer Electronics—Sensors "hide in plain sight" in many of the small devices that we carry with us or surround ourselves with on a daily basis. Smartphones, wearables, gaming devices, AR/VR systems, and IoT appliances absorb billions of sensors each year, driving increasingly context-aware, health-monitoring, and immersive experiences.

In many cases, small size and power limitations are the defining characteristics of the sensors used in these applications. The applications don't generally require the sort of precision that medical applications require but they are often significantly restricted on the power they can consume and the space within the overall package they can occupy.

To make the designer's life more interesting, there are frequently cost limits that are even more restrictive than the power consumption limits imposed by the overall design.

Aerospace, Defense, Energy, Logistics, and Mining—There are no more demanding environments for sensors than those presented in this market segment. Whether in outer space, deep mines, resource extraction wells, or military munitions, the sensors required by these applications must endure the harshest environmental factors humans have yet discovered.

From drone navigation, aircraft flight control, and grid optimization to mining equipment management—sensors deliver the environmental data that forms the basis for reliability, safety, and operational intelligence in mission-critical contexts.

While precision and accuracy are critical, physical packaging is often the most important factor in these applications, as it will allow the sensor to survive the environment in which it operates. In many of these applications, the anticipated operational lifespan of the sensor is quite limited, but the requirements placed on the sensor during that short operational life are rigorous.

In the same way that sensors used in medical applications must conform to rigorous specifications, aerospace, defense, and energy applications have standards and specifications (and occasionally multiple standards and specifications) to which they must conform. As with sensors intended for medical applications, the cost of meeting these specifications can be high and is typically reflected in the sensor's price.

TECHNOLOGY MEGATRENDS

Like so many other technology areas, sensor technologies are advancing at a rapid pace. Not surprisingly, AI is one of the ways in which sensors are evolving, and we'll look at that a bit later in this article. In addition, though, the following development pathways will define system design strategies involving sensors through 2030 and beyond.

Edge AI and On-Sensor Processing—Local, on-chip AI enables real-time signal analysis, decision-making, and even rudimentary inferencing, vastly reducing latency and bandwidth demands. Edge ML algorithms, running on sensor MCUs or dedicated AI accelerators, empower predictive maintenance, anomaly detection, and smart automation without requiring internet connectivity.

For engineers, this means moving data processing out of the cloud and into firmware or SoC-level architectures. System designs must accommodate the extra compute, memory, and energy management requirements for edge intelligence.

Miniaturization, MEMS & Nanomaterials—Sensors are becoming smaller, more sensitive, and energy-efficient due to the advancements in MEMS, advanced nano-materials (such as graphene and carbon nanotubes), and solid-state integration. These advances enable ultra-dense deployments, including wearables, implantables, environmental drifters, and remote sensor arrays.

Engineers building applications involving sensors should track advances in substrate materials, packaging, and heterogeneous integration to leverage the next leap in sensor performance and deployment density.

Sensor Fusion and Context Awareness—The fusion of data from multiple sensor types (motion, environment, biometric, vision, etc.) enables sophisticated context-aware systems. Multi-sensor modules, such as the DHT-11 shown in **Figure 3**, [4] and unified middleware facilitate higher reliability, accuracy, and smarter automation—essential for applications in autonomous vehicles, healthcare, and industrial IoT.

For fusion to occur, however, sensors must be network-aware and network-capable, allowing data to flow freely among sensor modules and between sensor modules and processing units within an application infrastructure.

Ultra-Low Power and Self-Generation—Energy-harvesting sensors (harvesting energy from sources that include ambient

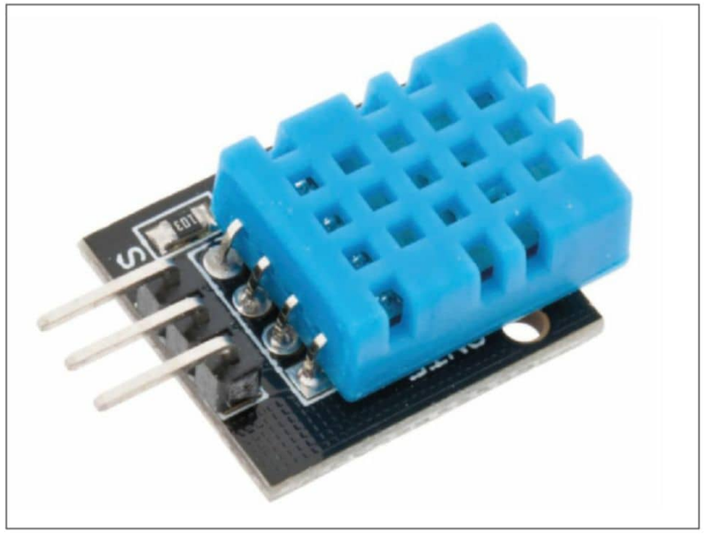


Figure 3
Multi-mode sensors present options for space-saving sensor deployments in challenging locations.

TECH FEATURE

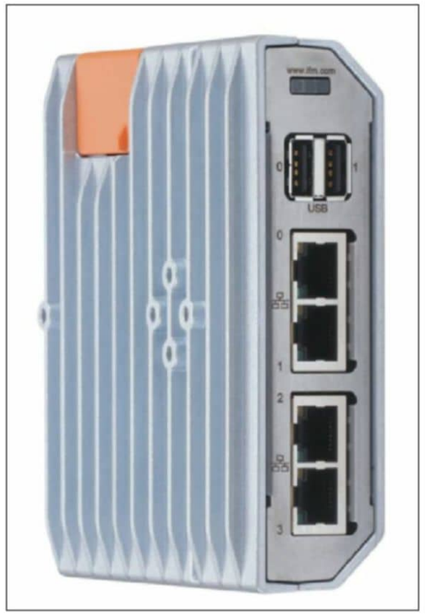


FIGURE 4
Sensor gateways are developing as rapidly as sensors themselves to make additional complex sensor applications more feasible.

light, vibration, and heat) and ultra-low-power MCUs pave the way for battery-less deployments. These units that require neither cabled power nor batteries that must be maintained and replaced are key for large-scale, maintenance-free sensor networks in cities, agriculture, and logistics.

Engineers must integrate power management, sleep modes, and energy harvesting circuits as core design elements, especially for remote and distributed deployments. While this set of considerations can free the designer from thinking about a handful of power considerations, it adds limitations on power consumption and the energy use profile to keep total solution power within the limits supported by the energy-harvesting sensors.

Advanced Materials and Quantum Sensors—Quantum and neuromorphic sensors promise orders-of-magnitude increases in sensitivity, resolution, and real-time AI compatibility. Applications for these advanced sensors will span navigation, imaging, process control, and high-security communications.

While not widely available for inclusion in current designs, designers should closely monitor emerging quantum and neuromorphic sensor SoCs, evaluating the chips, when released, for readiness in specialized applications

As with current computational AI, it is likely that there will be those who claim that neuromorphic and quantum sensors should be widely deployed in all but the most low-cost commodity applications. Engineers will have to carefully consider each case, though, to make sure that legacy technology isn't the most suitable and cost-effective solution for deployment in the field..

Wireless and Connectivity—New wireless standards—Wi-Fi 6/7, BLE 5.3, Ultra-Wideband—will drive ultra-reliable, low-latency sensor networks' These networks will support high-density mesh and self-organizing topologies critical for smart cities and Industry 4.0.

This connectivity is not something entirely in the future; current gateways such as the ifm AE2100 IIoT edgeGateway IP20 1S shown in **Figure 4** [5], are already making it easier to connect sensor networks to cloud analytics platforms and extend sensor-based applications across arbitrary distances.

Integration with cloud platforms, secure APIs, and remote management protocols is a must for future system designs. In these cloud integrations, engineers will need to look at the mean time between failure for remote sensors; replacing a low-cost sensor, can be an expensive process.

THE AI SENSOR REVOLUTION

Artificial intelligence is set to redefine embedded sensor networks at every level. While the physical basics of sensors will not change with the addition of AI, the processing of gathered data at the device level will change. That processing change will vary depending on the type of AI built into the solution and the uses to which that intelligence is put.

By 2030, AI-powered sensor nodes will make independent operational choices—reacting to change, local threats, and system health—without waiting on cloud processors. For application designers, this means developing distributed AI inference models, leveraging embedded ML frameworks, and ensuring hardware compatibility with their chosen AI engine (TensorFlow Lite, Edge Impulse, etc.).

The edge AI will be more critical for the application's success and will need to be available to the application on a consistent, high-availability basis.

Fortunately, current trends in technology development indicate that AI analytics will enable sensor networks to predict failures, schedule maintenance, and even self-repair (via network topology changes and task redistribution). Industrial sites, transportation grids, and smart cities will rely on self-healing mesh protocols and robust predictive ML pipelines in firmware. The predictive capabilities of AI will also benefit from diverse input from a variety of sensors.

AI is enabling advanced sensor fusion, which marries input from motion, vision, environmental, and user behavior sensors to create highly accurate, context-aware control systems. Rather than depending on single-stream sensor data, engineers should learn to architect their platforms to support multi-modal data ingestion, real-time fusion, and adaptive learning.

As richer data sets bring more capable decision-making at the network's edge they also bring greater possibilities for criminals to use the gathered information for malicious purposes.

With distributed sensor intelligence comes new cybersecurity risks. AI enables real-time anomaly detection, embedded encryption, and privacy-preserving computation—essential for critical infrastructure, medical devices, and personal electronics.

Security must be baked into hardware design (using technologies such as TPM modules and hardware roots of trust), firmware (with technologies that include secure boot and real-time threat analytics), and network APIs.

SENSING THE FUTURE

Sensor system design in the next decade will mean balancing the relentless advance of technology with operational realities: integration, energy use, security, interoperability, regulatory demands, and sustainability.

Engineers who embrace AI-driven architectures, modular integration, edge intelligence, advanced materials, and holistic security will lead the next wave of autonomous, context-aware, and future-proof sensor networks.

The next five years promise not just incremental upgrades, but fundamentally new paradigms of sensing, computation, and decision-making at the edge—the era of intelligent, secure, and sustainable embedded sensor systems is just beginning.

Additional materials from the author are available at: www.circuitcellar.com/article-materials

References [1] to [5] as marked in the article can be found there.

RESOURCES

Aosong Electronic Co., Ltd. | https://asairsensors.com

DwyerOmega | https://dwyeromega.com

ifm | https://www.ifm.com

McKinsey & Company | https://www.mckinsey.com

SICK | https://www.sick.com

DATASHEET

Datasheet: Small Size, Big Power

Smaller Microcontrollers Bring New Possibilities

By
Curtis Franklin,
Editor-In-Chief

A new generation of CPUs and GPus are bringing the sort of power once thought only possible for much larger systems to very small microcontrollers. These very small controllers offer OS, language, and application options that include AI and advanced communications to independent embedded applications.

Once upon a time, you could get a rough idea of how much compute power was available in a system by the size of the board holding the main processor. Big power required a big board.

In 2025, though, servers can sit in the box that once held strong mints and single-board computers can host serious AI. It is the age of the small microcontroller.

In many ways, the rise of the small controller can be traced to two parallel developments: The introduction of the Arduino ecosystem in 2005 and the release of the Raspberry Pi in 2012.

While neither of these was intended to be the sort of embedded controller used in massive production runs, they introduced many individuals to embedded system programming and redefined what could be expected from a very small embedded system controller.

It's not that tiny controllers weren't available before the early 2000s, it's just that tiny controllers came with minimal capabilities. That changed with these new families of boards, which meant limited size no longer came with sharply limited expectations.

So what defines a small embedded controller today? In this datasheet we're choosing to include boards that have multiple I/O capabilities, both analog and digital, and the ability to control those ports with complex logic.

That's the "embedded controller" piece, but the "small" part of the definition gets a little fuzzier. The Arduino Uno is 82.5mm × 53.3mm so that's one set of dimensions to throw around the size definition. Intel-based boards get small when they're in the 102mm × 102mm range, so there's that. And that's roughly where we decided to draw the lines—if a board fits inside that roughly 102mm × 102mm square, or just a bit more, then it's fair game for this list.

Once the size parameters are set, it's fair to ask about the capabilities that can fit on these diminutive single-board computers (SBCs). The capabilities start with the main processor on the board, which today is likely to be an Intel 13th-generation Core, an ARM-based processor from one of several manufacturers, a RISC-based AI-centric processor, or, increasingly, a combination of two of these.

The main processors are likely to be accompanied by a serious graphics processing unit (GPU) that can be deployed for either graphics or AI, or a neural processing unit (NPU) dedicated to on-board artificial intelligence.

These options are frequently accompanied by substantial RAM and enough on-board storage to allow for input data to be stored while it is used for programmatically controlled actions.

All the processing power is put to use through a wide variety of analog and digital input pins and data is shared over 10G Base-T networking, Wi-Fi, Bluetooth, cellular, and various serial communications protocols.

In short, size may be a more consequential factor than power when it comes to the modern small single-board computers featured here. They're capable enough for most embedded projects and provide a solid set of bridges to the AI-based embedded world that's just over the horizon.

NEXT MONTH'S TOPIC: Power-Monitoring Components Send product announcements to editor@circuitcellar.com

Ultra-Compact Intel

The AAEON de next-RAP8 is an ultra-compact SBC based on 13th-generation Intel Core processors. With up to 10 cores and 16GB of RAM, the de next-RAP8 requires 15W of power or less. The compute capabilities are coupled to the outside world through an interface package that includes dual 2.5GbE and 1GbE LAN ports, six USB ports, two RS-232/422/485 serial interfaces, and an 8-bit GPIO.

In addition to the on-board interface set, the de next-RAP8 is compatible with a variety of expansion adapters and modules that include AI acceleration, wireless communication, additional storage, and advanced video capture cards. These additional capabilities help make the board suitable for robotics, drones, and other robot-related applications for which it is marketed.

- 86mm × 55mm
- 13th Generation Intel Core i7/i5/i3 Processor
- Intel Iris Xe Controller
- TDP 15W
- Onboard LPDDR5x up to 16GB
- HDMI 1.4b × 1, eDP × 1
- 2.5GbE RJ45 × 1
- GbE RJ45 × 1
- USB 3.2 Gen 2 × 2
- USB 2.0 × 4
- M.2 2280 M-Key × 1 (PCIe [x2]), PCIe Gen 3 [x4] via FPC × 1
- 8-bit DIO/GPIO
- DC 12Vins

AAEON Technology
aaeon.com

A New OG

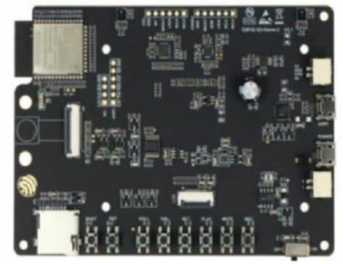
When Qualcomm announced the purchase of Arduino, the first results of the connection were announced at the same press conference. The Arduino Uno Q brings the two companies together on a single Arduino Uno-format board, with an Arduino STM32U585 microcontroller with ARM Cortex-M33, 2MB flash memory, and 786 kB SRAM to a Qualcomm Dragonwing QRB2210 processor featuring a quad-core ARM Cortex-A53 at 2.0GHz, Adreno GPU 3D graphics accelerator, and two image signal processors.

The UNO Q maintains full compatibility with a wide range of Arduino shields as well as libraries, sketches, and projects developed for the UNO ecosystem. UNO Q then adds a Linux-capable processor to allow a designer to run advanced applications while still controlling hardware through the STM32 MCU.

The new App Lab allows developers to orchestrate everything in one place, mixing sketches, Python scripts, and AI models.

- 82.5mm × 53.3mm
- Qualcomm Dragonwing QRB2210 processor:
  - Quad-core ARM Cortex-A53 at 2.0 GHz
  - Adreno GPU 3D graphics accelerator
  - 2 x ISP (13MP + 13MP or 25MP) at 30fps
- STM32U585 microcontroller:
  - Arm Cortex-M33 up to 160MHz
  - 2MB flash memory
  - 786kB SRAM
  - Floating Point Unit
- Bluetooth 5.1 with onboard antenna

Arduino
arduino.cc



Developing Video

Espressif's ESP32-S3-Korvo-2 board is intended for those who want to build video or graphics-heavy embedded applications based on the ESP32-S3-WROOM-1 microcontroller module. The ESP32-S3-WROOM-1 is a generic Wi-Fi + Bluetooth LE MCU module that is built around the ESP32-S3 series of systems on chips (SoCs). The SoC's ability to handle some AI-based tasks make it suitable for applications that include wake word detection, speech commands recognition, or face detection and recognition features in environments that include smart home, smart appliances, smart control panel, or smart speakers.

In addition to the microcontroller module the ESP32-S3-Korvo-2 board includes audio analog-to-digital conversion and Codec chips, twin microphones, and a power amplifier for audio output. There are also connectors for an available LCD and an external camera.

- 100mm × 68mm
- ESP32-S3-WROOM-1 microcontroller unit
- ES7210 four-channel audio-to-digital converter (ADC)
- ES8311 mono audio codec
- NS4150 EMI 3W audio power amplifier
- Twin on-board microphones
- USB power port
- Six function keys
- MicroSD slot
- FPC connector for LCD extension board
- External camera connector

Espressif Systems Co.
www.espressif.com

DATASHEET URLS:

AAEON Technology: https://www.aaeon.com/en/product/detail/de_next_de_next-rap8/specification

Arduino: https://docs.arduino.cc/hardware/uno-q

Expressif Systems: https://docs.espressif.com/projects/esp-adf/en/latest/design-guide/dev-boards/user-guide-esp32-s3-korvo-2.html

DATASHEET

Datasheet: Small Size, Big Power

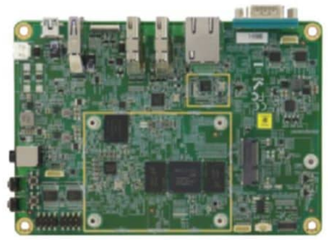
Tiny Industrial Linux

The OK153-S SBC from Forlinx Embedded is a compact industrial platform based on the Allwinner T153 processor. The Allwinner T153 integrates a quad-core ARM Cortex-A7 running at 1.6 GHz and a XuanTie E907 RISC-V core at 600MHz. Additional features include secure boot, hardware encryption, and multimedia output via RGB, LVDS, and MIPI DSI interfaces.

The OK153-S includes 256MB to 1GB of DDR3 RAM, along with 256MB or 512MB of NAND Flash and up to 8GB of eMMC storage. I/O includes three Gigabit Ethernet ports, two CAN-FD channels, and a Local Bus supporting 8-, 16-, or 32-bit operation up to 100MHz. There are also 10 UARTs, five I²C-compatible TWI buses, four SPI ports, and 24 analog input channels.

- 44mm × 35mm
- Allwinner T153 processor
- 256MB / 512MB / 1GB DDR3 RAM
- 256MB / 512MB NAND Flash
- Up to 8GB eMMC
- 2 × CAN-FD interfaces
- 2 × RS-485 with auto TX/RX control
- Wi-Fi + Bluetooth 5.0
- Mini PCIe slot for 4G modules
- 10 × UART, 5 × TWI (I²C compatible)
- 4 × SPI
- 3 × I²S (up to 384 kHz)
- 1 × DMIC (up to 48 kHz), 24 × GPADC
- 1 × TPADC (four-wire resistive touch)
- 30 × PWM/PWMCS
- 140 × GPIOs

Feiling Embedded Technology
forlinx.com



Small RISC-based AI

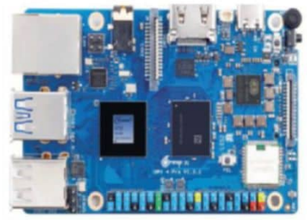
The IBASE Technology IBR500 is a RISK-based 147mm × 102mm single-board computer that features either a MediaTek Genio 700 (MT8390) or Genio 510 (MT8370) processor running at up to 2.2GHz. The IBR500 supports the processor with 4GB or 8GB LPDDR4 memory and 64GB or 128GB eMMC storage.

The IBR500 is intended for deployment in applications that prioritize AI capabilities, communications, or human user interface functions. The IBR500 supports dual display outputs with HDMI 2.0 and HDMI 1.4 at FHD60, and up to 3.2 TOPS AI inference capability. The board features GbE LAN, USB, and COM ports, as well as an M.2 E-Key expansion slot.

Supporting Yocto Linux, Android, and Ubuntu, the IBR500 provides developers with operating system flexibility.

- 147mm × 102mm
- MediaTek Genio 700 processor
- LPDDR4 4000MT/s on-board 4GB~8GB four channel 16-bits LPDDR4X at 3733MHz, up to 8GBs memory
- 64GB or 128GB eMMC 5.1 Flash for O.S.
- 1x HDMI (4K60) + 1x HDMI (4K30)
- ARM Mali-G57 MC3, OpenGL ES 1.1/2.0/3.2, Vulkan 1.0/1.1 graphics
- 1 × COM port, 1 × LAN port, 1 × USB 3.0, 1x micro USB OTG
- 1 × M.2 E-Key (2230) w/ SDIO, UART (for Wireless)
- 6 × GPIO in 2x pin header
- 1 × I²C in 2x 2-pin header
- 12V DC-in

IBASE Technology Inc.
ibase.com.tw



New Pro Pi

Orange Pi has released the Orange Pi 4 Pro, a compact single-board computer designed for high-performance edge applications. It integrates an eight-core Allwinner A733 processor with a 3 TOPS NPU, supported by up to 16GB of LPDDR5 memory. With the combination of compute capabilities, the Orange Pi 4 Pro combines AI acceleration with a wide range of expansion interfaces.

The Allwinner A733 processor combines two ARM Cortex-A76 cores and six Cortex-A55 cores clocked at up to 2.0GHz, paired with an Imagination BXM-4-64 GPU and a RISC-V E902 co-processor for auxiliary control. In addition, the on-board neural processing unit (NPU) delivers up to 3 TOPS of AI performance and supports INT8, INT16, FP16, and BF16 mixed precision operations. The NPU is compatible with popular frameworks such as TensorFlow, PyTorch, and ONNX.1

- 89mm × 56mm
- Allwinner A733 processor
- Imagination BXM-4-64 GPU
- RISC-V E902 co-processor
- Up to 16 GB LPDDR5 memory
- Optional eMMC modules: 16GB, 32GB, 64GB, or 128GB
- Optional SPI Flash: 128Mb or 256Mb
- Gigabit Ethernet, supports PoE power
- Integrated Wi-Fi 6 and Bluetooth 5.4 with BLE
- 40-pin expansion header (GPIO, UART, I²C, SPI, PWM)
- 1 × USB 3.0 host port
- 3 × USB 2.0 host ports
- 5 V DC input via Type-C connector

Orange Pi
orangepi.org

Compact Comms Power

The Particle Photon 2 is a small form-factor board that incorporates Wi-Fi networking and a pair of capable microcontrollers (a Realtek RTL8721DM MCU and an ARM Cortex M33 CPU) to present a tiny SBC with plenty of I/O that can sit in an edge deployment and communicate with the rest of the production infrastructure without requiring cables or significant hardware support.

In addition to the Wi-Fi networking, the Photon 2 provides Bluetooth Low Energy (BLE) using a single antenna for both, with an integrated connector for an external antenna, if needed. There is also up to 3MB of RAM available for user applications and a 2MB flash file system for application storage.

While the Photon 2 is capable on its own it contains the same compute module as the Particle P2, making the transition to an SMB mass-production deployment easier if needed.

- 36mm × 21.43mm
- Realtek RTL8721DM MCU
- ARM Cortex M33 CPU, 200MHz
- 3072KB (3 MB) of RAM
- 2MB flash file system
- 512KB static RAM (SRAM)
- 4MB pseudo-static RAM (PSRAM).
- 6 × ADC inputs
- 3 × UART serial interfaces
- 2 × serial peripheral interconnect (SPI) ports
- 1 × I²C port
- 1 × USB micro B connector

Particle Industries
particle.io

Arduino Architecture Plus

The Teensy 4.1, is a major expansion in the platform. It takes the capabilities of the Teensy 4.0 and expands them to be suitable for large-scale projects. .

The Teensy 4.1 adopts the larger Teensy 3.6 form factor, providing space for greater I/O capability. Additions include built-in 10/100 Mbit Ethernet, a microSD card socket, and USB host port.

Tightly Coupled Memory (TCM) allows the Cortex-M7 to use 512K of its 1024K total RAM as fast, single-cycle access memory via a pair of dual 64/32-bit high-speed buses).

The Teensyduino environment automatically allocates sketch code into this memory. This access is separate from the main AXI bus, allowing DMA-based peripherals to access other memory.

The processing speed of the Cortex-M7, combined with expanded memory makes the Teensy 4.1 a solid solution for projects that require high performance and connectivity.

- 61mm × 18mm
- ARM Cortex-M7 at 600MHz
- Floating-point math unit, 64 & 32 bits
- 7936K Flash, 1024K RAM (512K tightly coupled), 4K EEPROM (emulated)
- QSPI memory expansion, locations for two extra RAM or Flash chips
- USB device 480Mbit/sec & USB host 480 Mbit/sec
- 55 digital input/output pins, 35 PWM output pins
- 18 analog input pins
- 8 serial, 3 SPI, 3 I²C ports
- 32 general-purpose DMA channels

PJRC
pjrc.com

Very Small Pi

Featuring a higher core clock speed, double the memory, more powerful ARM cores, optional RISC-V cores, new security features, and upgraded interfacing capabilities, the Raspberry Pi Pico 2 delivers a significant performance boost while retaining compatibility with earlier Raspberry Pi Pico models. Raspberry Pi Pico 2 is a suitable microcontroller board for both enthusiasts and professional developers.

The Raspberry Pi Pico 2 is powered by an RP2350. The dual-core, dual-architecture capability of RP2350 allows users to choose between a pair of industry-standard ARM Cortex-M33 cores, and a pair of open-hardware Hazard3 cores. This gives developers a chance to experiment with the RISC-V architecture in a stable, well-supported environment.

This board supports the microcontroller with 520KB on-chip SRAM, 4MB on-board QSPI flash, 2.4GHz 802.11n wireless LAN, and Bluetooth 5.2. There are 26 multi-purpose GPIO pins, including four that can be used for ADC.

- 51mm x 21mm
- Dual ARM Cortex-M33 or dual Hazard3 RISC-V processors at 150MHz
- 520KB on-chip SRAM
- 4MB on-board QSPI flash
- 2 × UART
- 2 × SPI controllers
- 2 × I²C controllers
- 24 × PWM channels
- 3 × ADC channels
- 12 × PIO state machines

Raspberry Pi
raspberrypi.com

DATASHEET URLS:

Baoding Feiling Embedded Technology: https://www.forlinx.com/product/204.html

IBASE Technology: https://www.ibase.com.tw/english/download/RISC-Based_Embedded_Solutions_Catalog/IBR500_Datasheet.pdf

Orange Pi: https://www.orangepi.org/html/hardWare/computerAndMicrocontrollers/service-and-support/Orange-Pi-4-Pro.html

DATASHEET URLS:

Particle Industries: https://docs.particle.io/assets/pdfs/datasheets/photon-2-datasheet.pdf

PJRC: https://pjrc.com/teensy/IMXRT1060RM_rev3_annotations.pdf

Raspberry Pi: https://datasheets.raspberrypi.com/pico/pico-2-product-brief.pdf

Modern Embedded Insights

Designing Embedded Software Architectures That Last

By
Jacob Beningo

COLUMNS

I've reviewed hundreds of firmware projects over the years, and one thing always stands out: the most successful projects have a clear, deliberate architecture.

The rest start fast, look great for a few sprints, and then slowly collapse under their own weight. Sure, they have the right processes in place, and the developers have the right skills and tools, but they are missing something critical to their success: an architecture.

IEEE 1471 defines software architecture as "the fundamental organization of a system embodied in its components, their relationships to each other and the environment, and the principles guiding its design and evolution."

Architecture is the blueprint of what you're building. It's the roadmap that tells you how to get your system to the desired state. It's the decisions you wish you could get right early in a project.

Architecture isn't about drawing boxes and arrows—it's about clarity. Clarity of purpose. Clarity of flow. Clarity of design. In embedded systems, clarity is everything. It's what keeps firmware maintainable, testable, and scalable as products evolve.

Yet too often, teams skip architecture because they think it's "extra." It's not. It's the foundation that holds everything together.

For those who do think about architecture, it's not uncommon for them to think of it as the structure of their software. Over the years, I've learned that architecture isn't one-dimensional. It's not just structure. It's built on four interconnected parts which guide how we design and evolve software.

Today, we will explore the four critical pieces you must consider when designing a software architecture that survives the test of time.

In this article, we'll walk through:

- What software architecture is and why it matters
- Software Characteristics—what your system must be
- Design Philosophy—how you think about building it
- Architectural Decisions—why you chose a path
- Structure—how everything fits together

When you understand all four, you stop reacting to problems and start engineering solutions that last.

Let's dive in!

Note: These are not new ideas. Mark Richards and Neal Ford popularized them in their book Fundamentals of Software Architecture*. I've found that many electronics and embedded engineers are not familiar with them.*

WHY ARCHITECTURE MATTERS

Architecture defines the rules of engagement for your software. It shapes how the team thinks, communicates, and balances trade-offs.

In embedded systems, trade-offs are constant. Every decision, from choosing your Real-Time Operating System (RTOS) to selecting the right design patterns, affects performance, memory, and power. There's no perfect choice; there's only the best trade-off for your system's constraints based on your understanding of them at the time.

A good architecture helps you make those trade-offs intentionally. It connects decisions back to business goals, such as faster time to market, better reliability, and easier reuse. It also helps you connect your decisions back to the day-to-day implementation of software, ensuring that your software doesn't just work well today but is maintainable, flexible, and scalable for years into the future.

You can't perform those trade-offs in a vacuum while writing the software. You'll code yourself into a corner you'll never get out of.

I've often encountered teams that misunderstand Agile processes. They'll tell you that software architecture is emergent; you write your code, and, after many sprints, a beautiful, elegant architecture will naturally emerge.

Only one architecture I've seen emerges naturally from development: a giant ball of mud (**Figure 1**). It's an architecture that is tightly coupled, scales poorly, and even the slightest attempt to make a change results in a bug infestation.

Without an architecture, your project evolves by entropy. Each new developer adds code their own way. Interfaces blur. Data gets passed in ways that make sense to one person but confuses everyone else later. The result? Unpredictability.

Architecture keeps teams aligned. It's a shared language that allows developers to communicate design ideas clearly and adapt quickly when requirements change.

And, perhaps most importantly, it gives you a way to see your system: to visualize where data flows, where responsibilities live, and where complexity hides.

So, how do you define a software architecture?

There are many ways to define software architecture. The approach I like best is to look at defining your architecture as a puzzle (**Figure 2**), with four major pieces:

- Architecture Characteristics
- Design Principles
- Architecture Decisions
- Software Structure

Let's look at each of these pieces in detail.

SOFTWARE ARCHITECTURE CHARACTERISTICS

Architecture characteristics are the properties or traits your software must exhibit. These are the non-negotiables that define your system.

FIGURE 1
The most common software architecture: a big ball of mud

These aren't vague goals like "fast" or "scalable." They're concrete qualities such as:

- Reliability—The system behaves predictably, even under stress.
- Portability—Code can move across MCUs or compilers with minimal effort.
- Maintainability—The system can evolve without introducing chaos.
- Scalability—Features can grow without redesigning the foundation.
- Security—Data and access remain protected, by design.

Each of these characteristics has trade-offs. Reliability might come at the cost of performance. Scalability might require extra abstraction layers.

But defining them early gives your architecture direction. It sets the bar for design decisions. In fact, it helps set the tone for what design philosophies you should use, which then help guide your choices, which result in your software structure!

Whenever I start an architecture review, I ask teams to describe what "good" looks like for their system and then trace those goals to real design choices. Often, the teams struggling with bugs and other issues never defined these characteristics.

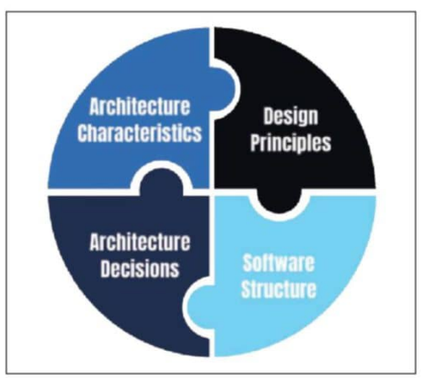


FIGURE 2
Software Architecture is like a four-piece puzzle.

COLUMNS

More than 70 different software qualities and characteristics might apply to your embedded system. I've listed my top favorites into typical categories that you can find in **Figure 3**.

This list is too much for anyone to trade off against. I recommend you choose the most important 5–7 characteristics and let those guide your way.

If you want to know the full list and go through my software characteristics worksheet to help you choose the right characteristics, a link can be found on the *Circuit Cellar* Article Materials and Resources webpage. [1]

DESIGN PHILOSOPHIES AND PRINCIPLES

At first, many engineers I talk with are tempted to consider design philosophies and principles an academic pursuit. Who has time to think about philosophy when there is code to be written and a system to be delivered?

Yet, a design philosophy is a practical set of ideas about designing a system that encompasses a set of guiding principles and practices that prioritize unique constraints and requirements.

For example, a critical principle for many software systems, including embedded systems, is that the software must be modular. Modularity helps us break the system into smaller, testable, and reusable pieces.

A design philosophy isn't a rulebook; it's a mindset—a practical way of thinking about constraints and trade-offs. It defines the way we think about how we want our embedded software to work.

Every byte, every cycle, every milliwatt must be earned. The right mindset turns those constraints into design clarity.

Over the years, I've refined a set of seven philosophies that I return to again and again when architecting embedded software. These are principles I use when I'm designing and building embedded systems but be warned, some are controversial to traditional firmware engineers:

- Data Dictates Design—Architecture emerges from understanding how data is produced, transformed, consumed, and stored.
- There Is No Hardware (Only Data)—Hardware is just an implementation detail. Abstraction layers are your friends.
- KISS the Software—Simple code wins. Clever code fails quietly (and sometimes wailing).
- Practical, Not Perfect—The real world rewards delivery, not elegance.
- Future-Proof for Flexibility—Today's hack is tomorrow's bottleneck. Plan for change.
- Quality Is a Mindset—Build in testing and validation from day one.
- Security Is Fundamental—Protect your data assets; they are your product.

Each philosophy is supported by actionable principles, the things you actually do in code.

For instance, "There Is No Hardware" leads directly to the Dependency Inversion Principle: high-level application logic depends on abstractions, not on specific drivers.

"KISS the Software" aligns with the Single Responsibility Principle: every module has one clear reason to change.

I see these philosophies as guardrails. They prevent the architecture from drifting into complexity or shortsighted optimizations.

Philosophy isn't theory. It's a practical lens that keeps embedded design sustainable.

ARCHITECTURAL DECISIONS

Architecture is full of trade-offs. You'll make hundreds of them over a project's lifetime. Some trade-offs are obvious, others subtle. What tools to use, how to partition tasks, and whether to use a service-oriented architecture or deployable containers are all architectural decisions.

Here's the problem I often encounter: After six months of development, you look back and can't remember why certain decisions were made! Even worse, sometimes a new team member joins the team and doesn't understand why things were done the way they were. Instead, they want to rip up everything that's been done simply because they think it's wrong when, in fact, certain trade-offs were made that actually make the architecture perfectly fine.

That's where Architectural Decision Records, or ADRs, come in.

An ADR is a short, structured note that documents one architectural choice: its context, the decision, rationale, and consequences. It's not a long report or a heavy process. Usually, it's a one-page Markdown file that lives alongside your source code.

Figure 4 shows what a typical ADR looks like. You can see that you don't have to go into super in-depth details. The idea is to capture the trade-off and why that direction best fits your project. It's also not set in stone. In the future, you can change your mind!

The point isn't bureaucracy—it's clarity. You're not writing for today's sprint; you're writing for the engineer six months from now trying to understand the "why."

Good ADRs are brief but complete. They explain trade-offs and assumptions without overwhelming detail.

Bad ADRs are either missing or so verbose they never get read.

If you're not sure whether to write one, here's the rule: if you had to think about why you made that architechtural choice, it's worth capturing.

Over time, ADRs become an architectural logbook, a breadcrumb trail of decisions that helps new engineers onboard faster and prevents old mistakes from being repeated.

I've seen teams cut onboarding time in half just by keeping their ADRs next to the code. When decisions are visible, architecture stops being tribal knowledge and becomes a shared discipline.

SOFTWARE ARCHITECTURE STRUCTURE

If you're reading this, congratulations! You're finally to the point of software architecture that you already knew about! Everything we've discussed before set the stage for defining the right structure for your architecture.

Once you've defined your characteristics, philosophies, and decisions, structure is what ties it all together.

Structure turns abstract thought into something visible, the map showing how the system fits together and where its boundaries are. This is where you get to draw block diagrams, arrows, UML diagrams, and all that stuff that shows the organization of the software.

One of the simplest and most effective ways to model that structure is the 4C Model, developed by Simon Brown. It organizes architecture into four levels of abstraction:

- Context—who uses the system and how it interacts with the outside world

Core System Qualities	Performance & Efficiency	Security & Safety	Design & Architecture	Operational & Behavioral	System & Deployment	Lifecycle & Business
Functionality	Performance efficiency	Security	Modularity	Predictability	Deployability	Cost efficiency
Correctness	Response time	Safety	Traceability	Stability	Installability	Time-to-market
Reliability	Throughput	Integrity	Low coupling	Precision	Updatability	Longevity
Robustness	Latency	Confidentiality	High cohesion	Accuracy	Recoverability	Toolchain sustainability
Fault tolerance	Jitter	Authentication	Abstraction	Responsiveness	Serviceability	Documentation quality
Availability	Resource utilization	Authorization	Separation of concerns	Synchronization accuracy	Diagnostic capability	Version traceability
Maintainability	Memory efficiency	Resilience to attacks	Simplicity	Observability	Field maintainability	Maintainability of processes
Testability	CPU utilization	Fail-safe behavior	Consistency	Restart capability	Remote diagnosability	Supportability
Modifiability	Real-time determinism	Compliance	Extensibility	Determinism	Rollback capability	Lifecycle traceability
Reusability	Power efficiency	Auditability	Flexibility	Responsiveness to faults	Firmware over-the-air (OTA)	Cost of ownership

FIGURE 3
Typical software characteristics for embedded systems

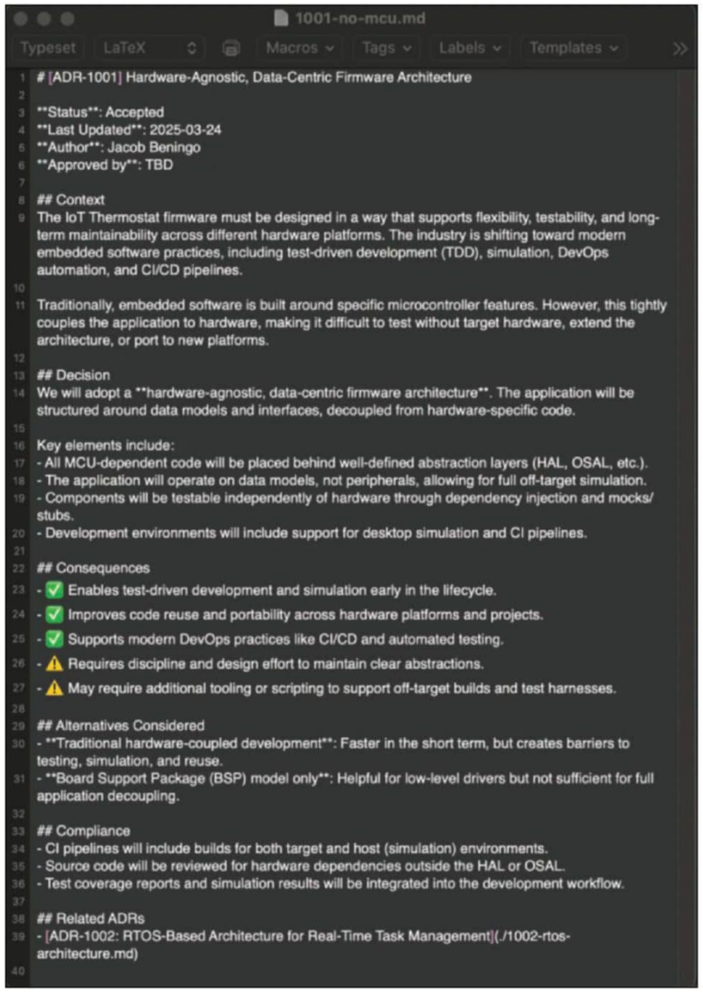

1001-no-mcu.md
Typeset
LaTeX
Macros
Tags
Labels
Templates
[ADR-1001] Hardware-Agnostic, Data-Centric Firmware Architecture
Status: Accepted
Last Updated: 2025-03-24
Author: Jacob Beningo
Approved by: TBD
Context
The IoT Thermostat firmware must be designed in a way that supports flexibility, testability, and long-term maintainability across different hardware platforms. The industry is shifting toward modern embedded software practices, including test-driven development (TDD), simulation, DevOps automation, and CI/CD pipelines.
Traditionally, embedded software is built around specific microcontroller features. However, this tightly couples the application to hardware, making it difficult to test without target hardware, extend the architecture, or port to new platforms.
Decision
We will adopt a **hardware-agnostic, data-centric firmware architecture**. The application will be structured around data models and interfaces, decoupled from hardware-specific code.
Key elements include:
- All MCU-dependent code will be placed behind well-defined abstraction layers (HAL, OSAL, etc.).
- The application will operate on data models, not peripherals, allowing for full off-target simulation.
- Components will be testable independently of hardware through dependency injection and mocks/stubs.
- Development environments will include support for desktop simulation and CI pipelines.
Consequences
- ✅ Enables test-driven development and simulation early in the lifecycle.
- ✅ Improves code reuse and portability across hardware platforms and projects.
- ✅ Supports modern DevOps practices like CI/CD and automated testing.
- ⚠️ Requires discipline and design effort to maintain clear abstractions.
- ⚠️ May require additional tooling or scripting to support off-target builds and test harnesses.
Alternatives Considered
- **Traditional hardware-coupled development**: Faster in the short term, but creates barriers to testing, simulation, and reuse.
- **Board Support Package (BSP) model only**: Helpful for low-level drivers but not sufficient for full application decoupling.
Compliance
- CI pipelines will include builds for both target and host (simulation) environments.
- Source code will be reviewed for hardware dependencies outside the HAL or OSAL.
- Test coverage reports and simulation results will be integrated into the development workflow.
Related ADRs
- [ADR-1002: RTOS-Based Architecture for Real-Time Task Management](./1002-rtos-architecture.md)

FIGURE 4
This is an example of an ADR, which is brief but explains trade-offs and assumptions for a specific code.

ABOUT THE AUTHOR

Jacob Beningo is an embedded software consultant and founder of the Embedded Software Academy, where he helps engineers master modern practices to design efficient, scalable embedded systems. He helps developers stay ahead in an evolving industry with hands-on training, expert insights, and strategies. He's consulted and trained companies such as Arm, General Motors, NASA, and others, completing more than 200 projects. Jacob has published books on embedded software development techniques. He holds three degrees, including a master's in space systems engineering from the University of Michigan. You can learn more about Jacob's work at www.beningo.com.

- Containers—the major executables or processes, such as your firmware app, RTOS tasks, or companion cloud services
- Components—the functional building blocks inside each container: drivers, middleware, services, and application logic
- Code—the source files and modules that implement those components

When applied to embedded systems, this model creates a consistent mental map for the whole team.

You start with a Context Diagram because it shows the users and your system at its highest levels. It shows all the big blocks that the user sees and that you need to account for. After all, aren't we building these systems for the users? It only makes sense that we start with them.

Figure 5 is an example context diagram for an IoT Thermostat.

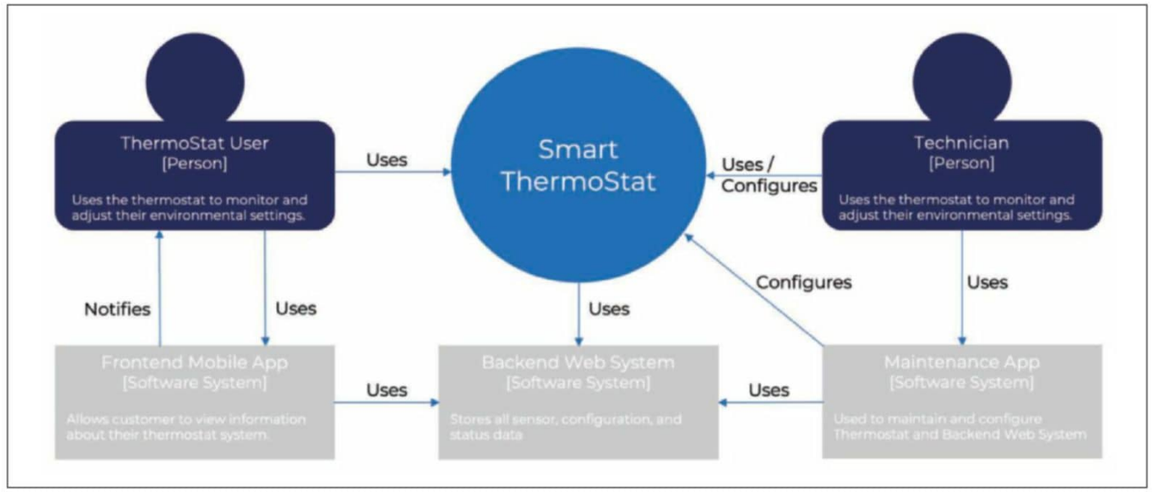


FIGURE 5
Context diagram for an IOT thermostat.

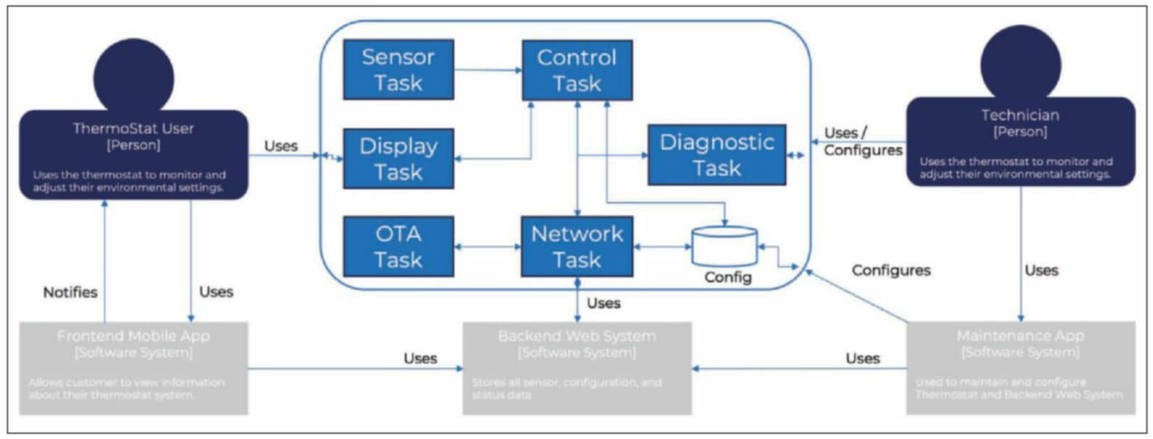


FIGURE 6
Container diagram for an IoT thermostat.

Once the context for this system is understood, we can then look at the containers. These could be literal containers like Docker that are deployed on a Linux-based embedded system. The containers could also be processes or threads that run on an RTOS. The purpose of the container diagram is to start to break the system up into the high-level structure.

At the same time, we keep the users in the picture so that we can understand the inputs and outputs of the system and how they affect the users (**Figure 6**).

Once we have our containers, we drill down into each. You can picture that each diagram goes into increasingly greater detail. Once inside the container, we can start to map out the components in the system that make up that container and how they interact with each other.

We analyze and dive deeper into every container as shown in **Figure 7**. While we are still drawing high-level blocks at this point, we might also draw service diagrams, sequence diagrams, flowcharts, cake diagrams, and so on to help fully explain the organization of the system.

My all-time favorite is using data flow diagrams, but we'll have to discuss that in the future. (Think back to the principles: data dictates design!)

Once you know the components that your containers need, you can design the code! You can use UML class diagrams, or you can just write the code and use a tool to generate the diagrams for you. Be warned, though: I haven't yet found any tools that do a great job drawing complex code diagrams.

An example code diagram can be seen in **Figure 8**. These are typically just UML class diagrams. Even if you are working in C, class diagrams can still be used to help you design modules, interfaces, and code structure you want to follow.

The beauty of the 4C model is that it's both structured and flexible. It scales from a single developer project to a multi-team product line.

But here's the catch—your architecture diagrams are only as good as your understanding of data flow. If you can't trace how data moves between containers and components, the diagram is just art. Follow the data through each level. It will tell you whether your architecture is balanced or brittle.

I often describe the sweet spot of modeling as the Goldilocks Zone: not too vague, not too detailed. Too little structure, and the diagram is meaningless. Too much, and no one maintains it.

Good architecture documentation captures just enough to communicate decisions, not recreate the source code. If you think you don't have the time to develop all these pictures, that's okay!

At least design your high-level architecture so you have the general map of what you are building, how it's organized, and how those components will interact with each other.

MANAGING EVOLUTION

Software architectures aren't simply nice to have, they're a necessity. They define what it is you are building and how it's organized.

You wouldn't build a house without a design, so what makes you think that building a software system without one is a good idea?

Don't let architecture fool you, though. Remember: it evolves! That means it's not static. You'll find over the course of any project that you'll need to update it, refine it, and evolve as you learn more and/or as requirements change.

At this point, you've got the tools to:

- Identify your software characteristics
- Define your philosophies and principles

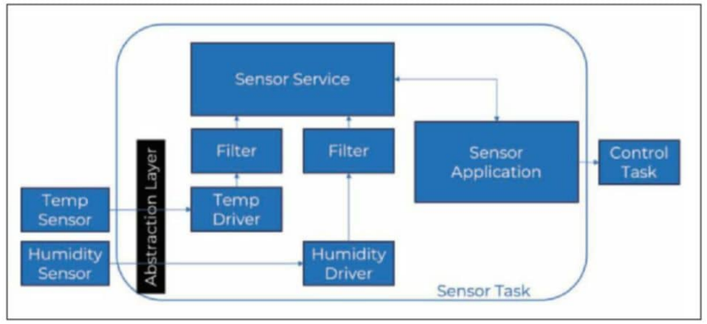


FIGURE 7
Component diagram for an IoT thermostat

- Document your decisions
- Decompose your software structure

Here's where you can go from here:

- Practice on a hobby project
- Apply these techniques to an existing project
- Start a new project from scratch

When you treat architecture as a living discipline, it becomes your most powerful asset.

You can change hardware without rewriting the firmware.

You can onboard new developers without halting progress.

And you can scale product lines without starting from scratch.

That's the goal: to design embedded software architectures that last.

So, the next time you start a project, don't jump straight into code.

Step back. Define your software characteristics. Write down your philosophies. Capture your decisions. Sketch the structure.

And most importantly: follow the data.

It will lead you to the architecture that's already hiding in your system.

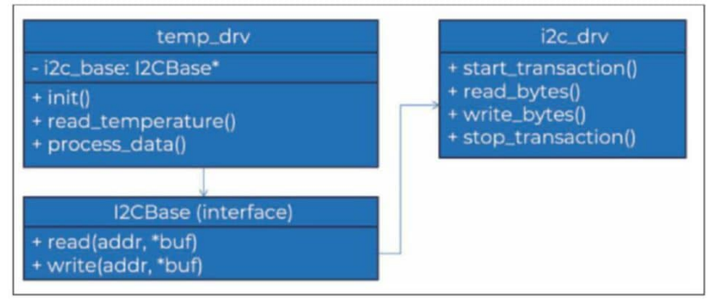


FIGURE 8
Code Diagram for an IoT thermostat, including a driver and abstract interface

Additional materials from the author are available at:
www.circuitcellar.com/article-materials
Reference [1] to [2] as marked in the article can be found there.

COLUMNS

Start to Finish

Understanding MCU Timing

What Actually Happens Inside That Chip?

By
Stuart Ball

What happens between your source code and the low-level machine code in your MCU is not usually obvious. In this article, Stuart investigates that disconnect.

COLUMNS

Most of the time when we work with an MCU, we design the hardware, write and debug the software, and then we're done. Long ago, when I started out in this industry, it wasn't uncommon to use assembly language. I wrote assembly code for both MCU and DSP designs. But it didn't take long for languages like C and C++ to replace assembly. Higher level languages provide a lot of benefits: type checking, memory management, simpler code, code that's easier to maintain, and at least some abstraction from the underlying hardware. There are C and C++ compilers for most MCUs, both free tools from the manufacturers and commercially available tools. But one thing you give up with a higher-level language is a connection between the source code and the related timing.

When I was in college, many years ago, I took about every programming course available, including COBOL (that tells you how far back that was, assuming you're old enough to even know what COBOL was). We were programming on an IBM 360 mainframe computer, and I discovered there was a compiler switch that would show you the underlying assembly code generated by the compiler. The printout (no interactive terminals, at least not for this class) would show you the line of COBOL code and then the underlying assembler statements. COBOL provides a high level of abstraction, so a single COBOL statement can produce many lines of assembly code. I turned in an assignment that way; what would have been a single page of COBOL statements was many pages of assembly code. The grader gave me 100 on the assignment but added a note: "Don't ever do this again." At that age, I thought it was humorous. The grader obviously did not.

We don't write code in COBOL anymore—at least I don't—but the same general principle applies to current high-level languages. A C compiler for an MCU is efficient at converting C statements to underlying machine code. But it's hard to get a good understanding of what that code does in terms of timing. And the higher the level of the language, the higher the abstraction, and the less you can predict what the underlying machine code looks like.

In most cases, you can safely write your code in C and not worry about it. If it compiles and runs and passes all the functional testing, you're good. But sometimes you want a little more detail about what's going on under the hood.

MCU TIMING EXPERIMENT

I became interested in this because I was working on a project using a character-based LCD. I've used them in previous *Circuit Cellar* articles; you may have used these, too. It's typically an LCD with a parallel or serial interface to an MCU, and it will display one to 8 lines of characters, from 8 to 32 characters in length. On LCD devices with a parallel interface, the MCU read/write timing has to match the capability of the LCD. Typically, the read/write strobe cycle time has to be about a microsecond long. That's not long enough to justify setting up a timer on most MCUs, but it can take many instructions to generate the timing. The general mechanism for doing that is something like this:

- Write ASCII to LCD data pins
- Set up read/write and command pins
- Set the LCD enable pin high
- Delay 500ns
- Set the LCD enable pin low
- Delay 500 ns

The "Delay 500ns" is usually implemented in a function, typically containing a string of assembly-language no operation (NOP) instructions or a loop that just idles for a bit. But I was curious as to how much of that 500ns time was taken up getting into the delay function. It's just a question of curiosity, but it has applicability when you need to know how quickly an interrupt service routine (ISR) is entered after an interrupt or how quickly you can start a function in the code.

Function Call Timing: **Figure 1** shows the schematic of a circuit I used for testing these ideas. The circuit is a Microchip ATMega324 MCU operating with an 18.432MHz crystal. The schematic is part of a larger development board. As I often do with these simple examples, I deleted from the schematic everything that's not relevant here. J1 is the programing connector; LED1 is a heartbeat LED that blinks once per second.

PB0, on test point TP1, is used to test the function call timing. TP1 is monitored with an oscilloscope set to capture a single transition. The code is in a loop that blinks the LED; once on every pass through the loop, it sets PB0 high then calls a function (named MegaFunction1) that resets PB0 and increments a dummy variable. **Figure 2** shows the result; it takes about 300ns from the time PB0 goes high until it's reset inside the function. That's about five MCU clocks—not too bad—but if it was used for timing the LCD enable function, that's about 60% of a 500ns delay.

The next experiment was to do the same thing but pass two parameters into a function named MegaFunction2. Since this requires that the MCU push the parameters onto a stack (probably a dedicated stack in this case), there is an extra delay for that process. **Figure 3** shows the result: it takes about 550ns for that call, almost twice as long as when passing no parameters. That's about 10 MCU clock cycles. If using this for LCD timing, you could call the function with two dummy

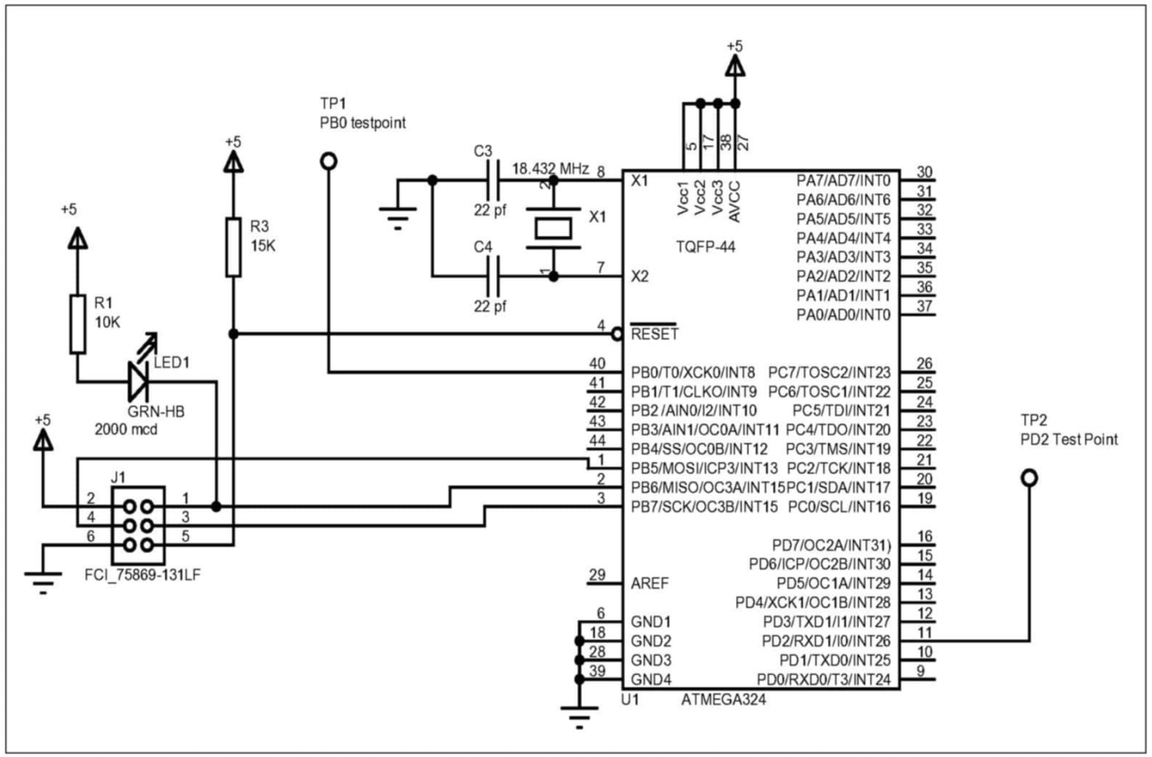


Figure 1
Schematic of the ATMega324 timing circuit

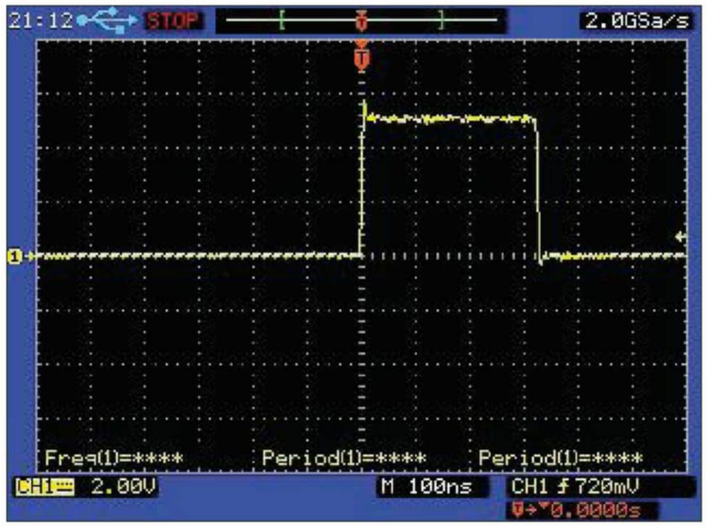


Figure 2
Function call timing, no function parameters.

parameters, do some useless thing with them, and return.

Interrupt Timing: I then wanted to see what it took to get into an ISR. The ATMega324 has three external interrupts shared with three port pins. The interrupt can be triggered even if the pin is configured as an output. I connected test point TP2 to port pin PD2, which is also the INT0 external interrupt. I configured INT0 to trigger on a rising edge, and I created an ISR to reset PD2 and increment a dummy variable. On each pass through the LED blink code, PD2 is set high, triggering the ISR. **Figure 4** shows the result; it takes over a microsecond to enter the ISR and reset the interrupt pin (note the 200ns timebase vs 100ns in the previous figures). If you were willing to dedicate an I/O pin to LCD timing, you could just generate an interrupt and return to time the LCD transfer.

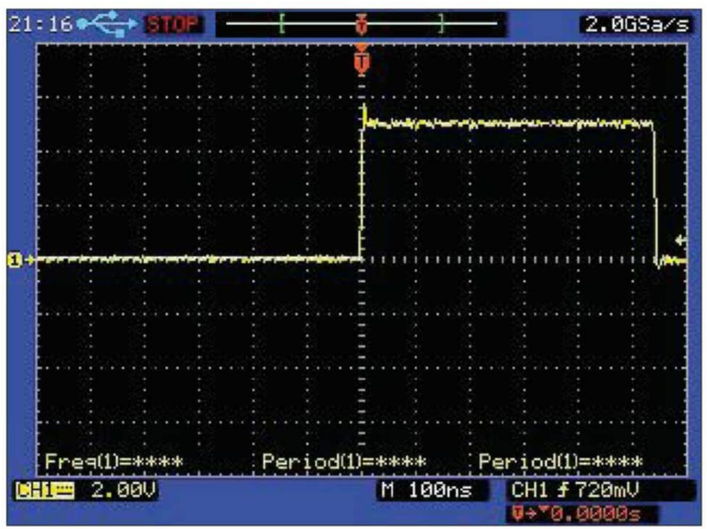


Figure 3
Function call timing, two function parameters.

Function Call in ISR: Usually you want an ISR to be as short as possible. But sometimes you need to make a function call inside an ISR. Depending on the compiler, it may have to save every register—the entire CPU context—to do this. So the effective ISR timing, with the added function call, may be much longer than just the ISR without a function call, and the function call itself may be longer than it would be outside an ISR. Keeping function calls out of ISRs is a good idea, unless you clearly understand what the compiler is going to do when adding that call. To demonstrate this, I added a call to MegaFunction1 inside the ISR but prior to clearing the PD2 bit. **Figure 5** shows the result (note the 1µs timebase); with the function call added, it is a full 3µs between the interrupt and clearing the interrupt. I didn't investigate in detail what was going on at the machine code level, but just adding a call to that simple function increased the time to clear PD2 by 3x. You may not have a choice whether or not to do this, but be aware that it can be costly in terms of performance.

MCU and Compiler: Of course, these timings are all based on one specific MCU, at one clock rate, using the Microchip Studio C compiler. I'd expect any ATMega device to have similar timings, ratioed to the clock frequency, of course. Other compilers might be more or less efficient in generating code for these functions. However, this shows that if you have performance requirements, something as simple as modifying a function to accept a parameter or two can affect the time it takes to start processing the function.

The long ISR completion time with a function call surprised me somewhat, but I'm sure that's a function of what context the compiler needs to save before servicing the interrupt. If you have very specific response time requirements for a project, it might be worth characterizing different compilers to see how it affects timings. Some compilers have performance settings that let you trade off code size or memory usage for speed.

Additionally, some compilers have a compile-time switch that will show you the assembly language output, like I did with the COBOL compiler I mentioned earlier. You can use that to produce assembler code and then go to the MCU datasheet to see how long each instruction takes. As it happens, Microchip Studio doesn't have this feature, at least not in the version I was using.

The source code for this demo is on the *Circuit Cellar* Article Materials and Resources webpage if you want to experiment with it.

TM4C1233D5

For comparison, I did the same tests using a TI TM4C1233D5 running at 80MHz (Figure 5). On the schematic, shown in **Figure 6**, W1 puts the TM4C into programming mode (short W1, press reset). J3 connects to an adapter and then to the PC serial port for programming. This is the same programming setup I've used in previous articles using the TM4C parts; I've included a schematic of the adapter in the files on the *Circuit Cellar* Article Materials and Resources webpage. As with the ATMega324 schematic, this is part of a larger development board, and I deleted everything not relevant here.

To save space, I didn't include the oscilloscope figures here, but a call to a simple (no parameters) function named MCUtiming1 resulted in 250ns to clear the bit (I again used PB0 of the MCU). With two 8-bit parameters in the called function, it took about 350ns.

The TM4C1233D5 at 80MHz is faster than the ATMega324 at 18.432MHz. Based just on clock speed, it should be about four times faster. In actuality, however, it is only about 20% faster for the first test and 36% faster for the second test. Some of this is explained by the way the flash memory works in the TM4C1233D5. The flash memory can be read at one word per clock cycle up to 40MHz CPU speed. But at 80MHz, the flash still runs at 40MHz. In addition, the TM4C1233D5 RAM has an odd/even addressing mechanism where successive read/write operations to the same bank will incur a single clock cycle delay. Still, if all were equal, the TM4C1233D5 would be expected to run about twice as fast as the ATMega324.

The code for this experiment was written using the TI Code Composer Studio compiler and the TI Tivaware libraries. The Tivaware libraries provide low-level drivers for the TM4C MCU hardware. These introduce an extra layer of abstraction as well as an extra call. Therefore, even writing to a single port using Tivaware can generate multiple machine code statements, if nothing else, the call/return to/from the Tivaware function.

To demonstrate this, I replaced the Tivaware call in the MCUtiming1 function with a macro that directly writes to the GPIO register. The resulting time to clear the bit went from 250ns to 150ns—half the time taken by the ATMega324. The point is that this experiment, in addition to using a different MCU, has a different toolchain as well. You can see that the performance compared to the AMTega324 is not what you would expect based solely on clock speed. The timing is based both on hardware and on the toolchain, illustrating the importance of the toolchain impact on performance.

TM4C Interrupt: I also conducted interrupt tests on the TM4C1233D5. The TM4C doesn't have the interrupt-on-output feature of the AVR, so I configured PB1 as an interrupt input, triggered on the rising edge, and I configured PB0 as the output to trigger the interrupt. The two pins are connected.

A simple interrupt with no internal function call results in PB0 being reset after 400ns—

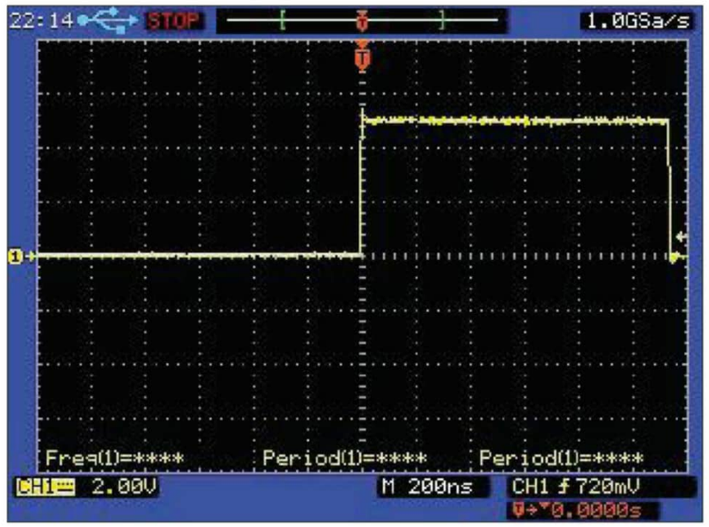


Figure 4
IInterrup timing example for the ATMega324.

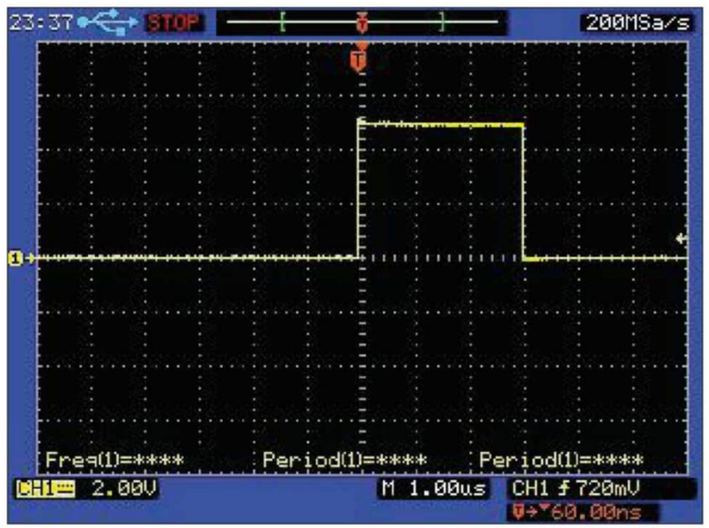


Figure 5
Interrupt timing with function call.

less than half of the AVR time—and in line with the fact that the TM4C is running at about 40MHz flash access speed, about twice the ATMega324 clock rate of 18.432MHz. Calling a function from within the ISR only extends the time to about 550ns, a much better result than obtained with the AVR. Thus, in this respect, Code Composer Studio for the TM4C1233D45 appears to be more efficient than Microchip Studio for the ATMega324. Again, different parts or different compilers could change this.

ROM Functions: The TM4C parts have an internal ROM that contains many of the Tivaware functions. The code can use these functions instead of placing them in the flash memory. I switched between the ROM-based and flash-based functions but did not see any improvement in the timing, although using ROM functions reduced code size about 20%.

Compiler Optimization: Code Composer Studio, like many compilers, has optimization features that will let you trade off between performance and code size. I set the optimization for maximum speed, but it made no difference in the timing. It also changed the code size very little. This doesn't mean that the optimization features don't work; it just means that this application is so simple, there isn't much to optimize.

Again, the code for this is on the *Circuit Cellar* Article Materials and Resources webpage.

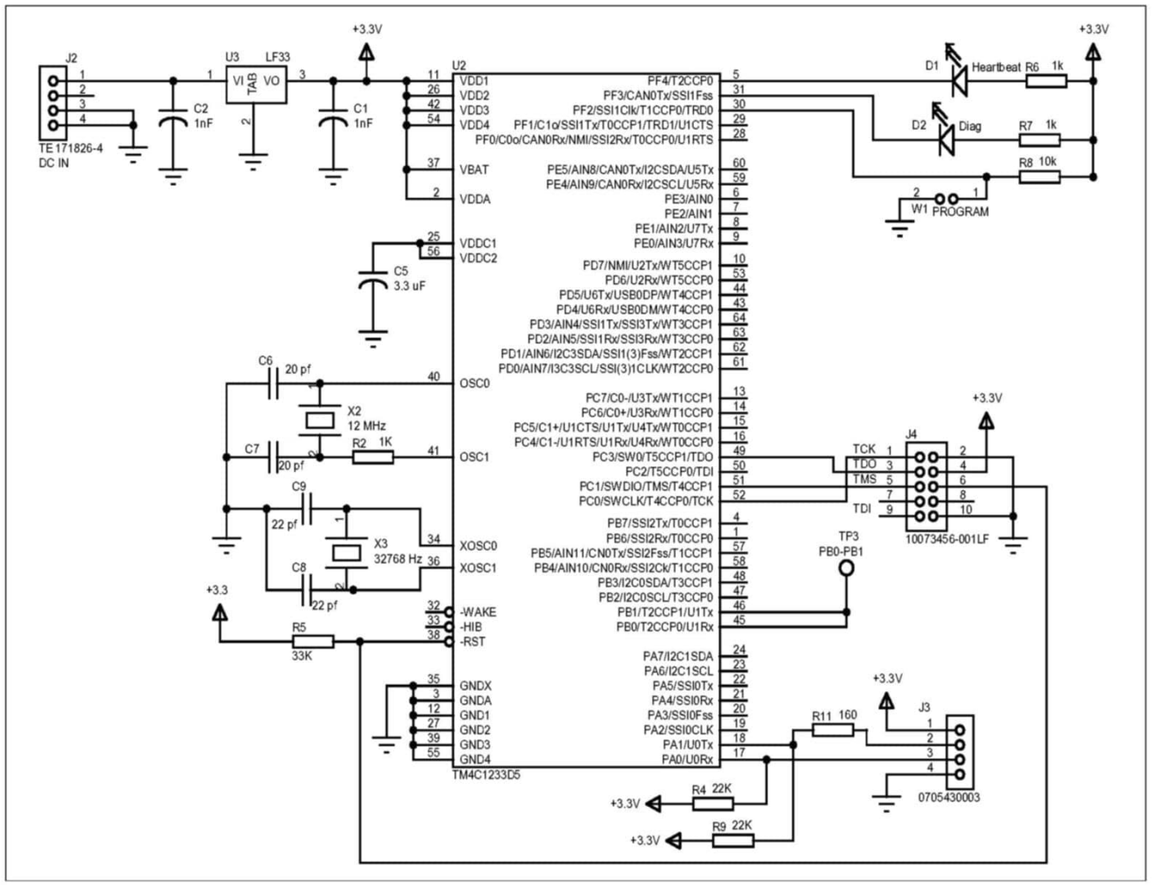


Figure 6
TM4C1233D5 timing circuit schematic.

and optimization. But optimizing MCU code is a more general requirement that applies to a wider number of situations.

Interrupt execution: In the TM4C example used here, there is a tick interrupt that occurs every millisecond. In this case, it just sets a flag that is serviced later, so the ISR execution time is always the same. But many years ago, I went with two other engineers to a field site to debug a problem that we hadn't been able to reproduce. An 8031 MCU was servicing an interrupt that, among other things, kept an internal counter related to machine operation. The problem was that a count would occasionally be skipped, and then it was off until power was cycled. It turned out that, under some rare circumstances, the ISR processing took too long and missed an interrupt. This wasn't a case where the MCU detected an error; the MCU went happily along, not even aware of what had happened.

To debug something like this, you could do what I did in these examples: dedicate a port bit to measure ISR timing. But in a more complex ISR, how do you know whether it's possible to miss an interrupt? The engineer who developed that 8031 code didn't know that the right combination of circumstances could cause that to happen. And it happened infrequently—just a few times per day—making it hard to track down.

Analyzing the code: The important thing about MCU performance is understanding what happens in the edge cases, especially with regard to what the toolchain actually produces in terms of machine code. Ordinary debugging tools, some of them anyway, can tell you how long the code spends in a specific function or ISR. But those tools can't anticipate the once-per-hour case where a worst-case stack up of delays occurs. In that 8031 example, the engineer never tested the worst-case scenario and possibly wasn't even aware it existed. And it all worked—until it didn't. Therefore, a large part of getting MCU timing right is understanding what those unexpected corner cases can do. This is especially important in equipment that can't be easily debugged. Think of a stick-on heart monitor such as the iRhythm Zio patch, a single-channel EKG device that a patient wears for a couple of weeks and then sends back for analysis. If something goes wrong because of unusual corner cases in software timing, there is no way to debug it remotely, which is a problem with many embedded systems. You can, of course, put debug information into EEPROM—assuming the software even has a way to detect that something went wrong. But even trying to test for worst-case outcomes requires some understanding of the circumstances that can create them.

JTAG: There are some debugging tools that can simplify timing tests; the ATMega324 and the TM4C series both have a JTAG (Joint Test Action Group) port that can be used for debugging. They require dedicated tools, and the JTAG pins aren't available for use in the application, at least while debugging.

Of course, the development system you use must support debugging via JTAG, and you need a JTAG debugger to make it work. Microchip supports debugging in their toolset, using the ATMEL-ICE debugger. TI has the XDS110 and CC-Debugger probes.

IAR systems has a development suite named Embedded Workbench. IAR has been around a long time. They have a JTAG debugger tool: the I-JET. Regardless of whether you use JTAG or a proprietary debug mechanism in the MCU, debugging capability is limited to that provided in the MCU.JTAG is a standard; JTAG debugging is kind of standard; but not every MCU supports the same debug features. All debuggers (all that I know about anyway) will let you stop at a breakpoint—kind of the most basic debugging feature there is.

Testing: Sometimes, testing is the only way to find these sorts of problems before they reach the field. Debugging the product may take more than just making sure everything works as expected; you may have to think about places where unexpectedly long code execution can create problems and test for those specifically. Whether you use a port bit, some custom debug output, or a JTAG connection, anticipating and resolving these cases before the product goes to the field may be time well spent.

HEADACHE AVOIDANCE

Although I've focused here on specific timing characteristics of two specific MCUs, the important takeaway is to understand what the toolchain does in converting source code to machine code and to anticipate and test worst-case scenarios to detect unexpected failures. Understanding the effects of toolchain abstraction on MCU performance can make the difference between something that works and something that mostly works but creates massive headaches somewhere down the line.

ABOUT THE AUTHOR

Stuart Ball recently retired from a 40+ year career as an electrical engineer and engineering manager. His most recent position was as a Principal Engineer at Seagate Technologies.

Additional materials from the author are available at:
www.circuitcellar.com/article-materials

Embedded in Thin Slices

Bob's Wrap Up

In Bob's last article with *Circuit Cellar*, he attempts to wrap up a career of more than 50 years as an embedded systems engineer and 14 years with *Circuit Cellar*. He looks at each of his 58 articles by category and provides some recommendations for his fellow engineers.

By
Bob Japenga

This month marks the end of the "Embedded in Thin Slices" column for *Circuit Cellar*. I am almost 10 years beyond my retirement from industry and I am a bit out of touch with this fast-moving field of embedded systems. KC, *Circuit Cellar*'s publisher, asked if I could end with a wrap up and farewell for my final article.

First, I want to thank everyone who was and is at *Circuit Cellar* for allowing me fourteen years to explore and share in print a few of the facets of my 50-plus years as an embedded systems engineer. I also want to thank all of you readers who have been so kind, encouraging me along the way with words of thanks for my articles. In a fast-moving field where we often feel behind the eight ball, I am always amazed that people actually can learn something from me.

I feel particularly blessed to have enjoyed my career and always feel for those who don't enjoy their professions as I have. Who would not want to have been part of landing a man on the moon (even if it was part of a cleanup effort after Apollo 17), or landing a probe on Mars (**Figure 1**), or flying around in corporate jets demonstrating your RNAV system (even when it misses the runway by 50 yards)? Who would not want to be part of bringing computer controls to a machine where so many operators lost fingers changing the settings on-the-fly on a music-box-style timing drum rotating at 20RPM (**Figure 2**) for a glass bottle manufacturer (**Figure 3**)? [1] I was particularly blessed the last 30 years of my career to work with a great team of embedded designers creating a wide variety of embedded systems for other companies.

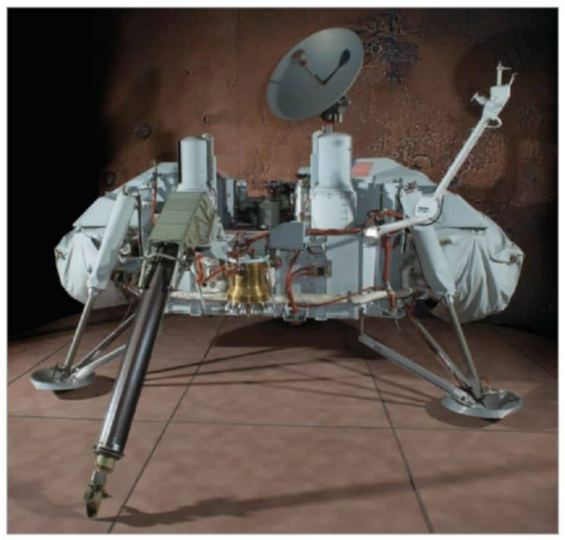


FIGURE 1
I was part of the Viking Mars Lander team.

I have provided in **Table 1** a comprehensive list of all 58 of the articles I wrote during my tenure at *Circuit Cellar*. If one piques your interest, why not obtain it from the *Circuit Cellar* archives for a nominal fee? [2] One good thing about getting older is that you can re-read something, and it seems brand new! Even if you yourself wrote it!

I have categorized the articles into seven disciplines, and now I will explain a bit about each discipline and how it impacted my career.

OPERATING SYSTEMS

Twenty-six of my 58 articles dealt with operating systems. I was trained as an aerospace engineer and graduated at the tail end of the Apollo program and the Vietnam War. In my junior year of college, I saw the job market for aerospace engineers tank (Apollo 18 was canceled in 1970) and therefore switched my focus to electronics while keeping my degree in aerospace engineering. Needless to say, I never had an operating systems course.

Operating systems are the glue that holds software systems together: scheduling tasks, interfacing with peripherals, and allocating resources. From the very beginning of my career, I was exposed to—and even made changes to—real-time operating systems like RT-11 and RSX-11S/M. A friend brought me a copy of Unix in the early 1970s, preparing me for my life with Linux years later. My first commercial software was written for a PDP-11 using RSX-11S to control a glass-bottle-making machine. In the late 70s, as we migrated away from the PDP-11 to the age of the microprocessor in our bottle-making controls, I left operating systems behind and ran everything from a simple *main* loop. Interrupts to the *main* loop provided concurrency and real-time control. Our number of peripherals and concurrent threads of operation were minimal, and things ran along fine until the late 80s. Our most complicated design without an operating system was an avionics box for the F-15. All were written in assembler, JOVIAL, or Ada.

As devices became more networked, our simple *main* loop became too cumbersome to manage the number of devices and resources we needed to control in real-time. It was time to go back to designs built on top of an operating system. The first was a real-time operating system called smx. A small and very fast little OS, smx helped us to control scores of devices and even some simple networks. Over the years, we used other OSs such as Nucleus, Integrity, QNX, and VxWorks.

We designed a major system (Time and Attendance) with Integrity and were throttled by a bug in the OS we couldn't fix without the source code. So, we ported the software to Linux. At the time, real-time extensions were not mainstream for Linux, but we were able to create a very responsive and reliable series of products enabling our customer to sell 100,000s of the devices. Over the next 20 years we incorporated Linux into many of the products we designed. It was nice when a customer needed to interface to a proxy to easily be able to provide that interoperability without writing any code.

Looking back 20 years, Linux was a mixed blessing. Interoperability was great, but keeping up with changes to the kernel and support for devices was a constant challenge. We were able to create some powerful systems, but what we saved in open-source free software, we paid dearly in one-time development costs. Saving the customer $30,000 per year for a license could get eaten up quickly in development costs. However, having a bug you couldn't fix was also costly. As Fred Brooks claimed, in software, there are no "silver bullets"—and there are no perfect OSs.

My recommendation: learn a few RTOSs—but only a few. Linux and FreeRTOS will cover a multitude of your needs.

SOFTWARE ENGINEERING

When I started out, software engineering was not even a discipline taught in schools. The Software Engineering Institute (SEI) at Carnegie Mellon wasn't founded until 1984. SEI has been a great resource for me in bringing engineering discipline to the realm of software design. In *Circuit Cellar*, I tackled software engineering topics such as concurrency, security, debugging techniques, filtering techniques, and requirements elicitation.

The engineering disciplines brought to software have enabled us to scale new heights. But, as I learned when I ventured into mobile development, the wonderful software we have underneath us still allows us to create powerful—yet undisciplined—designs without much structure, elegance, or maintainability. Even Flutter's attempt to enforce NULL safety can make your code more unreadable and unmaintainable when you follow their rules. I took several popular Flutter and React Native courses that taught the "how-tos" of coding without spending any time with the "whys." I fear our software is becoming more unreliable and less disciplined as our tools become increasingly powerful.

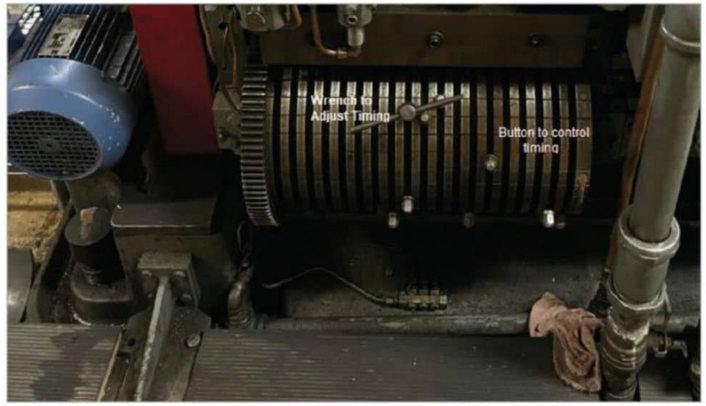


FIGURE 2
This glass bottle timing drum had a dangerous reputation; operators changing buttons while the drum rotated at 20RPM occasionally lost fingers in the process.

My recommendation to any of you who aspire to be great embedded software designers, is to learn the engineering disciplines for software. Learn how to practice information hiding as first proposed by David Parnas [3]; learn how to piecewise refine a design with good clean module partitioning [4]; learn how to write clear and concise software requirements [5]; learn how to write good documentation; and learn how to debug efficiently [6]. Those will be good starts—of course only in thin slices.

PROJECT MANAGEMENT

The first embedded software I wrote was a change to a PDP-11 controlling a glass bottle machine. I estimated it would take me three days, and it took me three weeks. It was a simple change. But learning how to estimate is an important part of managing a software

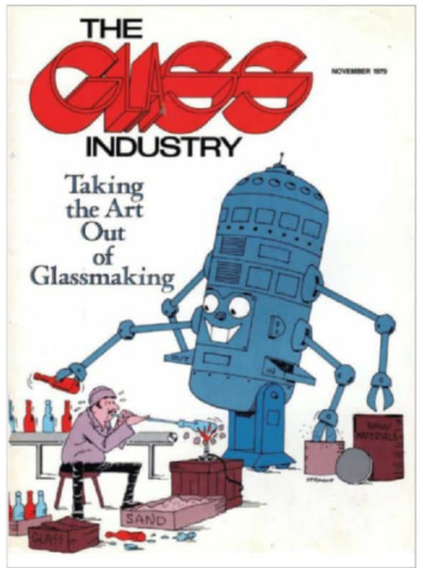


FIGURE 3
My cover story did not quite predict this outcome!

COLUMNS

project. Understanding and putting into practice Lehman's Laws of Software Evolution [7] will immensely help you manage your software projects. In addition, using checklists will help bring the software management behemoth under control [8].

INTERNET OF THINGS

In 1999, 25 years after I started my career, the term Internet of Things (IoT) was coined to describe a world of interconnectedness. We created our first IoT device in 2003—we were a little behind, you see—with an energy monitoring system that used a mesh network to talk to a server, which in turn connected to the Internet. It was clear that everyone and their neighbors were going to want to connect their devices to the web—both existing devices and new designs. This provided a significant amount of business for our company for the next 20 years. Each new request for proposal that came across my desk was wilder than the last. One that went on to be successful was a smart municipal garbage can that told the central office when it needed to be emptied [9]. Unfortunately, in the first design we didn't include a "odor" detector because we forgot what everyone knows: being full isn't the only reason to empty the trash!

I think there are still great opportunities for work in this area, and it's a great field to become an expert. There are more IoT devices now than there are people in the world. Global revenue from the IoT industry is expected to top $2.3 trillion dollars by 2030. It is where we put our efforts the last twenty years, and I don't see an end to growth in this area.

LOW POWER WIRELESS COMMUNICATION

If IoT continues to grow exponentially, most of the sensors are not going to be using WiFi or Ethernet to connect to the web. They will use low power, low cost, and generally low speed wireless communications. Although we toyed with Zigbee and proprietary mesh networks, in my 14+ years at *Circuit Cellar* I only wrote about two such technologies: LoRa and Bluetooth. Both are widespread and well-supported technologies I would heartily recommend every IoT engineer become familiar with.

MOBILE APP DEVELOPMENT

When you are over the hill and don't understand much of what your younger employees are talking about, why not learn a new technology? For the last two years, I have written about two frameworks for developing mobile apps that run on both iOS products and Androids: React Native and Flutter. The tools are amazing and powerful. A note to embedded engineers: No longer be afraid of having to create a mobile app to connect to your embedded device. Even I did. And it wasn't in thin slices.

THAT'S A WRAP

Thanks again to all of you readers who supported me writing about what I learned in my 50+ years as an embedded systems engineer.

Issue(s) and Date(s)	Category	Series Title	Article Titles
255, 257, 259, 261 October 2011 — April 2012	Operating Systems	Getting Started with Embedded Linux	When to Choose Linux for Your Embedded Design Choosing a Platform for Your Embedded Linux System Linux Licensing Issues Linux Software Development Tools
263, 265, 267, 269, 271, 273, 275, 277 June 2012 — August 2013	Software Engineering, Operating Systems	Concurrency in Embedded Systems	What it is and some of its pitfalls Concurrency, Atomicity and TOCTTOU Introducing Linux and Concurrency (4 articles) Using Files in Concurrent Linux Designs
279, 281, 283, 285 October 2013 — April 2014	Operating Systems	File Systems in Embedded Systems	File systems available on Linux for Embedded Systems (2 articles) Designing Robust Memory Systems (2 articles)
287, 289, 291 June 2014 — October 2014	Operating Systems	Configuring Linux	Configuring the Kernel Configuring Linux Applications Building Your Linux System
293, 295, 297 December 2014 — April 2015	Project Management, Software Engineering	Estimating Your Embedded System's Project	The Challenges of Estimating Software Projects The Challenges of Estimating Embedded Software Projects Approaches to Estimating Embedded Software Projects
299, 301, 303, 305, 307, 309, 311, 313, 315 June 2015 — October 2016	Internet of Things	The Internet of Things	Options for Connecting Wirelessly to the Internet Choosing a Wireless Carrier for Your Embedded System Connecting Wirelessly to the Internet with a small micro Cell Modem Certification Internet of Things Security (5 Articles)
317, 319, 321, 323, 325, 327, 329 December 2016 — December 2017	Project Management	How to Build an Embedded Consulting Company	Core Values of a Successful Company (2 articles) Axioms (2 articles) Marketing Your Company Trade-offs of Fixed-Price Contracts Writing the Proposal
331, 333, 335, 337, 339, 341 February 2018 — December 2018	Internet of Things / Software Engineering	Internet of Things Security	Command Injection Side Channel Attacks Secure Processors The Power of Checklists Identifying Assets Identifying Threats
343, 345, 347, 349 February 2019 — August 2019	Wireless Communication	Bluetooth Mesh	Alternatives Compared Provisioning Pondered Secure Provisioning Models and Re-use
351, 353, 355, 357, 359, 361, 363 October 2019 — October 2020	Wireless Communication	LoRa	Where does this Featherweight Fit In? The Things Network A Look at MachineQ Building a Prototype Device A MachineQ Example (2 articles) Building a Prototype LoRa Device
365, 367, 369, 371, 373, 375, 377 December 2020 — December 2021	Operating Systems	FreeRTOS	Intro to Real Time Operating Systems FreeRTOS Tasks Free RTOS Configuration FreeRTOS Inter-process Communication FreeRTOS Plus POSIX FreeRTOS Security FreeRTOS, TCP and the Internet
379 February 2022	Software Engineering	Lehman's Laws of Software Evolution	Rules to Code By
381, 383, 385, 387, 389, 391, 393, 395, 397 April 2022 — August 2023	Software Engineering	Debugging Real Time Systems	The Bugs Bugs of Time Bugs the Bug Us Tools to Help Us Debug Tools Need to Debug on the Go Strategies to Determine if a Problem is in HW or SW Strategies for Duplicating Bugs Flawed Assumptions in Finding Bugs Strategies for Finding Bugs
399 October 2023	Software Engineering	How Thin Slicing Can Help Us	The Search for "Fists" in Embedded Systems
401, 403 December 2023 — February 2024	Software Engineering	Eliciting Software Requirements for Embedded Systems	Conventional Methods An Agile Approach
405, 407, 410, 411, 413, 415, 417, 419 April 2024 — June 2025	Mobile Development	An Embedded Designer Looks at Mobile App Development	Flutter and React Native React Native Eco Systems React Native Development React Native Hooks Managed Development (Expo) for Releases Asynchronous Programming in Flutter Introduction to Flutter State Management Releasing Your Flutter App to the App Stores
421 August 2025	Software Engineering	Input Filtering Algorithms for Digital Signal Processing	Extracting Signals from Noise and Other Distractions
423 October 2025	Software Engineering	Fixed Point Arithmetic	

TABLE 1
This catalog lists all 58 of my "Embedded in Thin Slices" Articles in *Circuit Cellar*.

TABLE 1
Continued

Additional materials from the author are available at: www.circuitcellar.com/article-materials
References [1] to [9] as marked in the article can be found there.

ABOUT THE AUTHOR

Bob Japenga has designed embedded systems since 1973. From 1988-2020, Bob led MicroTools, an engineering firm specializing in creating a variety of real-time embedded systems across a wide range of industries. Bob has been awarded 11 patents in many areas of embedded systems and motion control. Now retired, he enjoys building electronic projects with his grandchildren. You can reach him at Bob@ListeningToGod.org.

COLUMNS

TDK Announces 1U High, 13-Output Modular Power Supply Delivering up to 1500 W with Extremely Low Acoustic Noise

TDK Corp. announces the expansion of the TDK-Lambda MU series with the introduction of the MU6 modular power supplies. Featuring extremely low acoustic noise and BF-ready medical isolation, the MU6 can provide up to 1200W with an input of 85V AC to 264V AC (derating below 90V AC) and 1500W with a 180V AC to 264V AC input in a compact footprint. Up to 13 fully isolated output voltages can be easily configured using TDK-Lambda's online Quick Product Finder. Applications include medical, dental, broadcast, professional audio, test and measurement, and industrial equipment that require a low-profile 41mm height and/or low audible noise.

The MU6's patented intelligent fan speed control circuit uses a microcontroller algorithm to monitor not only the temperature of the primary converter, but also each of the individual secondary modules. This enables the fan speed to be accurately adjusted to provide an optimum audible noise level as low as 36dBA, even with an 80% load. This low acoustic noise level improves hospital patient comfort and reduces operator fatigue.

An extensive range of modules is available from 150W to 1008W, providing continuous coverage from 3.3V to 105.6V DC. Each module includes remote sense and an output good signal as standard. Options for the MU6 include a 5V/2A or 12V/1A standby voltage, fan fail signal, inhibit and enable for all outputs, AC fail, PMBuscommunication, and current share for configurations requiring parallel operation. Providing enhanced flexibility to meet diverse system requirements and simplify the design process.

The MU6 power supplies deliver efficiencies of up to 94% meaning less energy is lost as heat, resulting in lower operating temperatures and increased reliability. These units can operate continuously in ambient temperatures ranging from -20°C to +70°C, with the load linearly derated to 50% above 50°C. The earth leakage current is less than 300µA, including all tolerances, with a touch current of <100µA. A market-leading seven-year warranty is provided as standard, delivering unmatched long-term assurance of product reliability and support, and reinforcing the highest standards of customer confidence in the industry.

The MU6 has an input to output isolation of 4000V AC (2x MoPP), an input to earth isolation of 1500V AC (1x MoPP), and an output to earth isolation of 1500V AC (1x MoPP), which is generally required for B- and BF-rated medical equipment. Output-to-output isolation of 1500V AC (1x MoPP) is available for enhanced safety and streamlined medical system design.

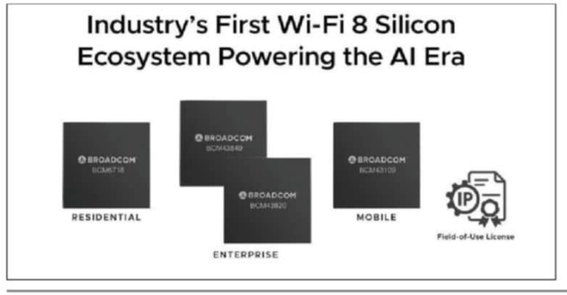
TDK Corp. | us.lambda.tdk.com

Broadcom Introduces Industry's First Wi-Fi 8 Silicon Ecosystem Powering the AI Era

Broadcom, Inc. unveiled the first Wi-Fi 8 silicon solutions for broadband wireless, targeting residential gateways, enterprise access points, and smart mobile clients. These solutions are designed for the performance, reliability, and efficiency needs of AI-driven edge networks. To accelerate AI-first wireless connectivity, Broadcom also offers Wi-Fi 8 IP licensing for IoT, automotive, and mobile devices.

The new family of Wi-Fi 8 solutions include:

- BCM6718 designed for residential and operator access applications
- BCM43840 and BCM43820 tailored for enterprise access applications
- BCM43109 for edge wireless clients such as smartphones, laptops, tablets, and automotive



Combining advanced scheduling technology with Wi-Fi 8 capabilities, Broadcom's Wi-Fi 8 solutions ensure faster speeds, longer range, optimized spectrum use, reduced contention, and predictable performance even in challenging operating environments. Broadcom's Wi-Fi 8 silicon future-proofs the network with breakthroughs in ultra-low latency, intelligent traffic management, and multi-device performance management at scale.

Broadcom's Wi-Fi 8 access silicon includes a hardware-accelerated telemetry engine, a game-changer for AI-driven network optimization. This engine collects real-time data on network performance, device behavior, and environmental conditions, serving as a critical input for AI models. This telemetry can be leveraged by customers to train and run inference on the edge or in the cloud:

Wi-Fi 8 marks a shift from chasing raw peak speed to redefining connectivity through Ultra High Reliability (UHR)—delivering consistent, predictable network performance even under challenging conditions.

For the AI edge, an open, scalable, and power-efficient network is essential, with AI-aware devices working together. Broadcom's Wi-Fi 8 solutions offer a smaller footprint and better battery efficiency than Wi-Fi 7. To foster rapid scaling for the AI era, Broadcom will license its Wi-Fi 8 IP, creating an AI-aware ecosystem for IoT, automotive, and mobile devices that complement AI-first access points.

Broadcom, Inc. | broadcom.com

Analog Devices Launches ADI Power Studio and New Web-Based Tools

Analog Devices, Inc. (ADI), a global semiconductor leader, announced the launch of ADI Power Studio, a comprehensive family of products that offers advanced modeling, component recommendations, and efficiency analysis with simulation. In addition, ADI is introducing early versions of two new web-based tools with a modernized user experience under the Power Studio umbrella: ADI Power Studio Planner and ADI Power Studio Designer. These new tools, together with the full ADI Power Studio portfolio, streamline the entire power system design process. The Power Studio tools support engineers from initial concept through measurement and evaluation, empowering engineers to design with confidence and efficiency.

Today's electronic systems require more power density than ever, with dozens or even hundreds of power rails and interdependent voltage domains. That complexity creates bottlenecks and requires rework during architecture decisions, component selection, and validation. Power Studio addresses these challenges by providing a unified, intuitive workflow that helps engineering teams make better decisions earlier by simulating real-world performance with accurate models and automating key outputs.

ADI Power Studio Planner: The next-generation, web-based tool for system-level power tree planning gives engineers an interactive view of their system architecture, providing clarity to model power distribution, calculate power loss, and analyze system efficiency with ease.

ADI Power Studio Designer: A powerful, web-based tool for IC-level power supply design provides optimized component recommendations, performance estimates, and tailored efficiency analysis. Built on the trusted ADI power design architecture, offering guided workflows so engineers can set key parameters and move confidently toward simulation, configuration and evaluation.

Power Studio Planner and Power Studio Designer are available now as part of the ADI Power Studio. These tools represent the first phase of ADI's vision to deliver a fully connected power design workflow for customers, with ongoing updates and product announcements planned in the months ahead.

Analog Devices, Inc. | analog.com

GD32 MCU Family Expands High-Performance Portfolio with New GD32F503/505 Series MCU

GigaDevice, a leading semiconductor company specializing in Flash memory, 32-bit microcontrollers (MCUs), sensors, and analog products, announced the official launch of the GD32F503/505 series of high-performance 32-bit MCUs, expanding its product range based on the ARM Cortex-M33 core. These new MCUs are built upon the ARM Cortex-M33 core, offering enhanced performance and features designed for a variety of applications. The GD32F505 operates at a frequency of 280MHz, while the GD32F503 runs at 252MHz, providing a solid foundation for demanding tasks in areas such as digital power supplies, industrial automation, motor control, and robotics. Samples are currently available, with mass production slated to commence in December.

The GD32F503/505 series excels in performance, achieving up to 4.10 CoreMark/MHz and 1.51 DMIPS/MHz, thanks to the efficient Armv8-M architecture and advanced hardware accelerators. Memory configurations include up to 1024KB of Flash and 192KB of SRAM, allowing developers to flexibly allocate memory resources based on their specific application needs. This flexibility is crucial for optimizing performance and cost-effectiveness in complex designs.

The MCUs are equipped with a comprehensive set of peripherals, including three ADCs with a sampling rate of up to 3Msps, multiple communication interfaces such as SPI, I²C, UART, and CAN, and a robust timing system. This set of peripherals facilitates seamless connectivity and precise control in various applications. The series is engineered for reliability, operating within a voltage range of 2.6V to 3.6V and supporting an industrial-grade temperature range of -40°C to +105°C. They also incorporate robust ESD protection, withstanding up to 15kV air discharge, ensuring long-term stable operation in harsh environments.

Security is a key focus, with a comprehensive chip-level security architecture that includes a Secure Boot and Secure Firmware Update platform, along with hardware security features for data protection. Multi-layered security mechanisms, including multi-channel watchdogs and hardware CRC, ensure system reliability and data integrity. Furthermore, the series has obtained German IEC 61508 SC3 (SIL 2/SIL 3) functional safety certification. The devices also incorporate a built-in hardware security engine integrating SHA-256 hash algorithms, AES-128/256 encryption algorithms, and a True Random Number Generator (TRNG).

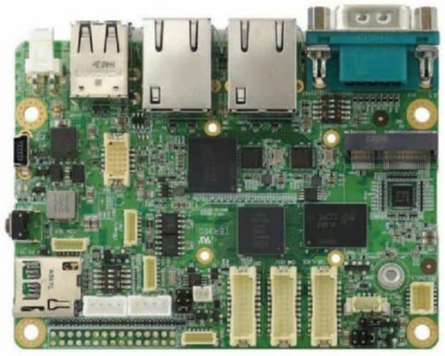
The GD32F503/505 series offers a range of package types, including LQFP, QFN, and BGA options, to accommodate various compact device designs. Comprehensive documentation, including datasheets, software libraries, and ecosystem guides, is available to support developers, making it easier to integrate these MCUs into their projects.

GigaDevice Semiconductor, Inc. | gigadevice.com

Compact IBR300 2.5” SBC Powered by NXP i.MX 93 from IBASE

IBASE Technology, Inc., a leading provider of rugged embedded computing platforms, announced the release of the IBR300, a 2.5” RISC-based single board computer (SBC) powered by the NXP i.MX 93 processor with dual-core ARM Cortex-A55 (up to 1.7GHz) and a Cortex-M33 MCU. Designed for embedded and industrial edge deployments, the IBR300 delivers powerful processing and real-time control in a compact, low-power form factor optimized for space-constrained systems.

Featuring 2GB LPDDR4 memory, 32GB eMMC storage, and an SD socket for easy expansion, the IBR300 provides a complete embedded solution for space-constrained applications. A comprehensive set of industrial-grade I/O interfaces, including an M.2 E-Key (2230) slot for Wi-Fi/Bluetooth modules, ensures flexible connectivity for factory automation, smart retail, machine control, and IoT gateway deployments. Most notably, the IBR300 supports a wide operating temperature range from -40°C to +85°C, delivering stable and reliable performance in harsh or fanless environments while minimizing maintenance and integration complexity.



The IBR300 empowers developers to accelerate time-to-market for intelligent devices and edge systems requiring efficient performance, compact design, and rugged reliability. It exemplifies IBASE’s ongoing innovation in ARM-based embedded computing, complementing its portfolio of industrial motherboards, edge AI systems, and IoT platforms.

Compact and Versatile: The 2.5” form factor simplifies integration into existing systems and custom enclosures, reducing footprint and development time.

Dual-Core Performance with Real-Time Support: Combines application-class Cortex-A55 cores with real-time Cortex-M33 for both complex workloads and deterministic control.

Wide-Range Operating Temperature (-40°C ~ +85°C): Ensures dependable operation in extreme industrial conditions, ideal for fanless and outdoor systems.

Flexible I/O and Expansion: Built-in memory, storage, SD socket, and an M.2 slot provide a ready platform for customization and rapid deployment.

Edge and Embedded Focus: Designed for intelligent edge nodes, smart devices, and networked machines requiring low power and high reliability.

IBASE Technology, Inc.
www.ibase.com.tw

Alif Semiconductor Elevates Generative AI at the Edge with New Support for ExecuTorch Runtime in Its Ensemble MCUs

Alif Semiconductor, the leading global supplier of secure, connected, power efficient Artificial Intelligence and Machine Learning (AI/ML) microcontrollers (MCUs) and fusion processors, announced that developers can now use the ExecuTorch Runtime, a quantization extension of the popular PyTorch ML framework, for AI applications built to run on its Ensemble E4/E6/E8 series of MCUs and fusion processors.

The ExecuTorch extension enables developers of AI models aimed at resource-constrained hardware targets, such as MCUs, to use the PyTorch framework to create lightweight models that support low-latency, high-accuracy inference. Since the Ensemble E4, E6, and E8 products include the ARM Ethos-U85 NPU, which supports transformer-based ML networks, developers can now use PyTorch and the ExecuTorch Runtime to build generative AI applications to run locally in battery-powered endpoint devices such as smart glasses, human-to-computer interfacing, healthcare and diagnostics, robotics, transportation, toys and education, smart homes, and smart city equipment.

At the PyTorch Conference (Moscone West, San Francisco CA, October 22-23), ARM demonstrated examples of generative AI applications compiled with ExecuTorch and running on Alif’s Ensemble E8 fusion processor. A small language model running on the Ensemble E8 was capable of generating stories suitable for children in response to visual prompts, and extendable to verbal prompts.

Real-time on-device speech-to-text models was suitable for integration into wearables such as smart glasses. The demo performs real-time transcription of speech to enable live captioning.

Alif Semiconductor | alifsemi.com

NEW PRODUCT SUBMISSIONS— E-mail: product-editor@circuitcellar.com

IDEA BOX

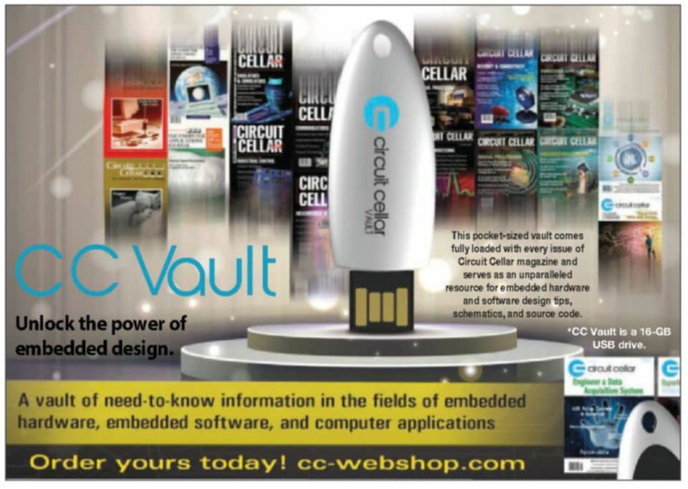
The Directory of PRODUCTS & SERVICES

AD FORMAT:
Advertisers must furnish digital files that meet our specifications (www.circuitcellar.com/mediakit).

All text and other elements MUST fit within a 2" x 3" format.
E-mail adcopy@circuitcellar.com with your file.

For current rates, deadlines, and more information contact Hugh Heinsohn at 757-525-3677 or Hugh@circuitcellar.com

When it comes to robotics, the future is now!
Advanced Control Robotics simplifies the theory and best practices of advanced robot technologies, making it ideal reading for beginners and experts alike.
With this book, you'll learn about:
• Communication
• Technologies
• Control Robotics
• Embedded Technology
• Programming Language
• Visual Debugging...
and more
ADVANCED CONTROL ROBOTICS
HANNO SANDER
Get it today, cc-webshop.com


CC Vault
Unlock the power of embedded design.
This pocket-sized vault comes fully loaded with every issue of Circuit Cellar magazine and serves as an unparalleled resource for embedded hardware and software design tips, schematics, and source code.
*CC Vault is a 16-GB USB drive.
A vault of need-to-know information in the fields of embedded hardware, embedded software, and computer applications
Order yours today! cc-webshop.com

embeddedTS
embeddedTS.com
TS-7100-Z
Powered by the high performance, ultra-efficient Arm® Cortex®-A7 based i.MX6UL CPU
...In an enclosure as small as two decks of cards
(2.4" x 3.6" x 1.7")
Best In Show
Embedded Computer Design
Made in the USA
with global parts

TEST YOUR EQ

Contributed by David Tweed

Answer 1 —

Lee key difference between AWGN and quantization noise is the PDF, or Probability Density Function, which is a description of how the values (voltage or current levels in analog systems, or data values in digital systems) are distributed.

The values from AWGN have a bell-shaped distribution, known variously as a Gaussian or Normal distribution. The formula for this distribution is:

$$f(x,\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

μ represents the mean value, which we take to be zero in discussions about noise. σ is known as the "standard deviation" of the distribution, and is a way to characterize the "width" of the distribution.

It looks like the image in Figure 2:

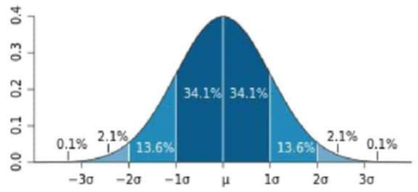


FIGURE 1
Source: Wikipedia

While the curve is nonzero everywhere (from −∞ to +∞) it is important to note that the values will be within ±1 σ of the mean 68% of the time, within ±2 σ of the mean 95% of the time, and within ±3 σ of the mean 99.7% of the time. In other words, although the peak-to-peak value of this kind of noise is theoretically infinite, you can treat it as being less than 4σ 95% of the time.

On the other hand, the values from quantization noise have a uniform distribution — the values are equally probable, but only over a fixed span that's equal to the quantization step size of the converter. The peak-to-peak range of this noise is equal to the converter's step size (resolution).

However, it's important to note that both sources of noise are "white", which is a shorthand way of saying that their effects are uniformly distributed across the frequency spectrum.

Answer 2 —

The power of a noise signal is proportional to the square of its RMS value.

The RMS value of AWGN is numerically equal to its standard deviation.

The RMS value of quantization noise is simply the peak-to-peak value (the step size of the converter) divided by √12, or V_{RMS} = 0.2887 VP-P. This is easily derived if you characterize the quantization noise signal as a small sawtooth wave that gets added to the analog signal.

Answer 3 —

When combining noise sources, you can't simply add their RMS voltage or current values together. From one sample to the next, one noise source might partially cancel the effects of the other noise source(s).

Instead, you add the individual noise power levels to come up with an overall noise power level. Since power is proportional to voltage (or current) squared, this means that you need to square the individual RMS measurements, add them together, and then take the square root of the result in order to get an equivalent overall RMS value.

$$V_{RMS(total)} = \sqrt{V_{MRS}(n1)^2 + V_{RMS}(n2)^2 + \ldots}$$

Answer 4 —

As described in the previous answer, uncorrelated noise sources are added based on their power. With AWGN, the noise in one "segment" of the frequency spectrum is not correlated with another segment of the spectrum, so if you have a particular voltage level of noise in, say, a 1-Hz band of frequencies, you'll have √2 times as much noise in a 2-Hz band of frequencies. In general, the RMS noise level for any particular bandwidth is going to be proportional to the square root of that bandwidth, which is why the devices are characterized that way.

So, if you have an opamp that's characterized as having an input equivalent noise level of 2 μV/√Hz, and you want to use this in an audio application with a bandwidth of 20 kHz, the effective input noise of the opamp will be 2 μV × √20000, or about 283 μV_{RMS}. If your input signal is a sinewave with a peak-to-peak value of 1V (353 mV_{RMS}), you'll have a signal-to-noise ratio of about 124 dB.

Student SUBSCRIPTIONS!
When textbooks just aren't enough, supplement your study supplies with a subscription to Circuit Cellar. From programming to soldering, robotics to Internet and connectivity, Circuit Cellar delivers the critical analysis you require to thrive and excel in your electronics engineering courses.
Sign up today and
Sign up today and SA
Sign up today and SAVE 50%
50% OFF
www.circuitcellar.com/subscription
circuit cellar
MCU-BASED COLOR DATA ACQUISITION

The Future of Sensors in Safety Systems

Sensing the Stop

How Magnetic Sensors Are Enabling the Next Generation of Braking Systems

By
Karsten Köhler,
Manager Business Development, TDK-Micronas GmbH

We initiate a complex and highly coordinated process every time we press the brake pedal in our vehicles. That familiar action relies on a sophisticated system that converts motion into control, ensuring safety at every stop. Today's braking systems must function seamlessly under a variety of conditions, whether halting a high-speed SUV or easing a car into a tight parking space.

Magnetic sensors have emerged as a critical component in making these capabilities possible. Their ability to provide accurate, real-time information about mechanical position, motion, and force enables responsiveness and intelligence within braking architectures. As vehicles become more electrified and software-defined, these sensors are increasingly important.

THE HIDDEN COMPLEXITY BEHIND EVERY STOP

Modern vehicles must be capable of coming to a stop quickly and safely under enormous loads. For instance, a high-powered SUV accelerating from 0 to 100 kilometers per hour in five seconds can travel over 140 meters in that time. To stop safely in less than a quarter of that distance, the braking system must dissipate nearly a megawatt of kinetic energy while maintaining stability and steerability.

This same system must also perform nuanced actions like holding the vehicle steady on a hill or gradually slowing to a stop in heavy traffic. Braking systems depend on high-performance mechanical hardware supported by smart electronic control to achieve power and precision. Accurate sensor input is essential for determining component positions such as pedal angle, piston movement, and brake pad engagement.

THE SHIFT FROM HYDRAULIC TO ELECTRONIC BRAKING

For decades, braking systems have relied on hydraulic circuits to transmit force from the pedal, such as the model shown in **Figure 1**, to the wheels. However, the increasing adoption of electrification is reshaping this architecture. Traditional systems are being augmented, and in some cases replaced, by electromechanical solutions.

The introduction of electro-hydraulic boosters and electric parking brakes marked the beginning of this shift. These systems reduce complexity by replacing vacuum components and mechanical cables with compact, electronically controlled motors. Hybrid configurations, where the front wheels retain hydraulics for higher braking loads while the rear wheels use electronic actuators, are already on the road today.

In the near future, brake-by-wire systems will become increasingly common. These systems eliminate hydraulic fluid altogether and replace it with electrical signaling and motor-driven actuation. The result is a cleaner, lighter, and more controllable system, especially critical for electric vehicles and autonomous platforms.

HOW MAGNETIC SENSORS SUPPORT BRAKING CONTROL

The transition from hydraulic to electronic braking depends on reliable sensing. Magnetic sensors operate without physical contact and are particularly well-suited to automotive environments due to their resistance to vibration, temperature extremes, and contamination.

Magnetic sensors are used in multiple ways throughout the braking system. They monitor how far the driver presses the brake pedal, enabling the ECU to understand intent and apply braking force accordingly. They also track the position of pistons and calipers, confirming whether braking mechanisms are fully engaged or retracted in locations such as the one illustrated in **Figure 2**. In systems where hydraulic fluid is still used, sensors help detect low fluid levels in the reservoir, triggering warnings or adaptive responses. Additional applications include confirming whether the parking brake is set and integrating with wheel speed data to support features like hill-hold assist.

SPECIALIZED SENSOR TYPES FOR SPECIALIZED ROLES

Braking systems rely on multiple sensor types, each engineered for specific tasks. Linear sensors track horizontal movement, such as the distance a pedal travels. Rotational sensors detect angular position changes, essential for electric motors controlling brake

FIGURE 1
Brake pedal assembly with sensor and actuator module. The sensor detects drivers' intent.

FIGURE 2
Brake caliper with actuator modulet. The sensor detects brake piston travel, ensuring drivers' intent is applied.

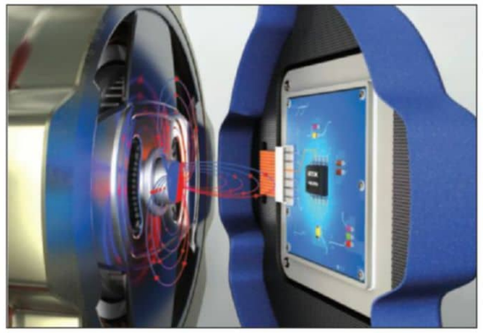


FIGURE 3
Detailed view of the actuator module. The sensor controls the turns of the shaft of the electric motor by sensing the poles of the magnet at the end of the shaft, translating into the travel of the piston.

calipers or boosters. Some applications use 3D or multi-axis sensors to capture compound motion, such as a switch being pressed and rotated simultaneously. Even simple magnetic switches, which detect whether a component is open or closed, remain relevant for end-position detection.

The variety of sensor functions reflects the complexity and precision required in braking applications. No single sensor type can handle every role, so systems are designed with a combination of specialized sensors, each selected for its performance and environmental tolerance.

ADDRESSING EMC CHALLENGES AND ENSURING REDUNDANCY

The rise of electrification brings a new challenge: electromagnetic compatibility. With electric motors, inverters, and other high-current components generating magnetic fields throughout the vehicle, sensors must be able to distinguish meaningful signals from background interference. Techniques such as differential signal processing, advanced sensor geometries, and embedded shielding are used to isolate the target magnetic field from environmental noise.

Safety is another critical factor. Braking systems must comply with standards such as ISO 26262, which require systems to function even in the event of a component failure. To meet these requirements, sensor redundancy is essential. Importantly, this redundancy must be heterogeneous, meaning using two different sensing principles or technologies to ensure that a single mode of failure cannot affect both. This design philosophy enhances fault tolerance and supports the overall safety architecture of the vehicle.

REAL-WORLD EXAMPLES OF SENSOR INTEGRATION

Magnetic sensors can be found at every critical control point in a modern braking system. Near the brake pedal, angle or multi-axis sensors capture the driver's input with high resolution. This information allows the system to apply braking force proportionally and precisely.

Within motorized actuators, such as those used in electromechanical brake systems, sensors monitor shaft or gear position to confirm accurate application and release of braking force. These sensors provide real-time feedback to the control unit, allowing for quick adjustments and consistent braking behavior.

Multiple sensors may be deployed in parallel to ensure system reliability, often with each sensor using a different measurement principle. For example, a Hall-effect sensor may be paired with a tunnel magnetoresistance (TMR) sensor to create a redundant, fail-safe arrangement.

In high-EMC environments like electric or hybrid vehicles, sensors with differential signal architectures, such as the system shown in **Figure 3**, can maintain accuracy without heavy shielding. These advanced designs reject common-mode noise, focusing only on changes in the target field.

THE ROAD AHEAD: SOFTWARE-DEFINED BRAKING SYSTEMS

The move toward brake-by-wire mirrors a trend seen in steering systems, which have transitioned from hydraulic to fully electronic control. As braking systems evolve in the same direction, the number of magnetic sensors used per vehicle is expected to grow. In some designs, more than 20 sensors may be deployed, each fulfilling a specific function in the braking control chain.

This increased sensing capability opens the door to enhanced vehicle performance. Braking systems will not only stop the vehicle but also adapt to road conditions, predict wear on components, and integrate more deeply with autonomous driving features. With the right sensor infrastructure, software-defined braking can deliver both safety and adaptability.

SMART BRAKES

Magnetic sensors have moved from supporting roles to central pillars in the architecture of modern braking systems. Their ability to deliver precise, resilient, and real-time data is critical to everything from pedal feel and caliper control to system diagnostics and predictive maintenance.

As electrification and automation reshape the automotive landscape, magnetic sensing will continue to underpin safe, intelligent, and responsive braking, ensuring that every stop is as smart as it is secure.

TECH THE FUTURE

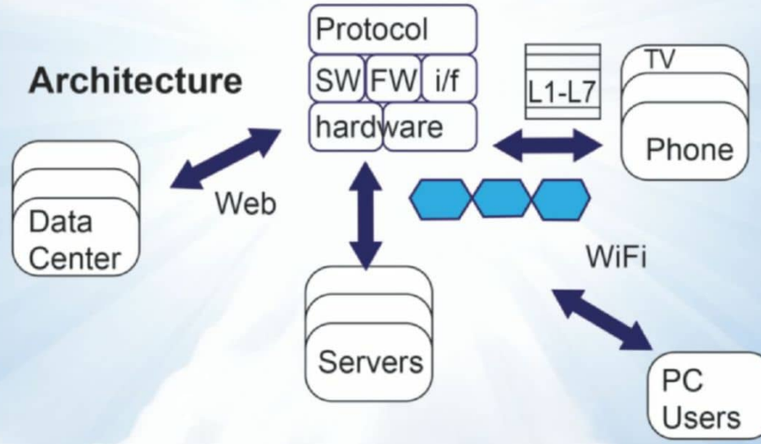
RESOURCES

TDK-Micronas
www.micronas.tdk.com

ABOUT THE AUTHOR

Karsten Köhler works as a Business Development Manager at TDK-Micronas in Freiburg, Germany. He is focusing on application of semiconductors, mainly sensors and drivers, in automotive applications. With more than 20 years of experience in leadership roles in electronics technology development, sales / key account management, and product management, he is blending technological innovation with sustainable business models. Karsten is taking active roles in industry associations, e.g., ZVEI Germany, and standardization gremiums. He is publishing regularly on the role of semiconductors in automotive innovations, such as electrified drivetrains, steer-by-wire, or brake-by-wire.Karsten holds master's degrees in engineering as well as business administration.

SeaFire® Micros, Inc.
IP & Technology for Cloud Networks



Design through Production
Military — Commercial

Hardware (Analog/Digital)
Firmware • Software • FPGA

Patented Network Technology | Unique Cloud Technology
No Compete Government Procurement
No Compete Government Funding

20% DISCOUNT on 1st Purchase

info@seafire.com | seafire.com

TS-9370

Low Power, High Performance SBC

The TS-9370 is a versatile and power-efficient industrial SBC built for rugged applications. Powered by the NXP i.MX9352 64-bit ARM® architecture plus two Cortex®-A53 processors operating up to 1.7 GHz, it delivers robust performance with low power consumption.

32 GB eMMC (upgradable)

2 GByte LPDDR4 RAM

MikroBUS socket
For MikroE Click Boards

Made in the USA
with global parts

embeddedTS

TS-4100

Ultra-Efficient and Fully Featured

The TS-4100 features an NXP i.MX 6UltraLite CPU, industrial temp range, FPGA, eMMC, and customizable I/O. Comes preloaded with Debian Stretch.

TS-7100-Z

As Small as Two Decks of Cards

Powered by an Arm® Cortex®-A7 i.MX6ul CPU, it runs efficiently at <2W, featuring Dual Ethernet, Bluetooth, WiFi, and NimbeLink/Digi sockets.

www.embeddedTS.com